

**Specification and Design of
Real-Time Distributed Systems**

**Design-Level Specifications of
Selected Aspects of SHINPADS**

M. M. Higginson
J. David Andrews
Andyne Computing Limited
Kingston, Ontario

June, 1988

DSS Contract File: 05SV.W2207-6-AF53
Serial: W2207-6-AF53/01-SV



Français

Contact Us

Help

Search

Canada Site

Home

Defence R&D Canada

Defence Site



Top



Previous

Document: 1 of 1



Next



Bottom



Order



Download

Browse Indexes

Titles

Authors

Descriptors

Corporate
Authors

Corporate
Acronyms

Document
Numbers

Search Database

Summary

TITLE: SPECIFICATION AND DESIGN OF REAL-TIME DISTRIBUTED
SYSTEMS: DESIGN-LEVEL SPECIFICATIONS OF SELECTED ASPECTS
OF SHINPADS

PERSONAL AUTHOR: Higginson, M.M.; Andrews, J.D.

CORPORATE AUTHOR: Andyne Computing Ltd, Kingston ONT (CAN); Chief
of Research and Development, Ottawa ONT (CAN) Director of Research and
Development Communications and Space

DESCRIPTORS: Specifications; Design; Real time operations; Computer
programs; Software development; SHINPADS; Distributed
systems; MASCOT; Computer networks

REPORT DATE:
15 Jun 1988

NUMBER OF PAGES : 98

DRDKIM N°:
90-04503

CANDIS N°: 66736

Format(s): Hardcopy; Originator's fiche received by
DSIS; Document Image stored on Optical Disk

Date modified: 2005-04-05

Important Notices

It is recommended you
search "Authors" and
"Corporate Authors" on
their respective "Browse"
pages.

Click here if you are having
problems displaying text (ie.
text is too big, or too small)
or page appears to be
jumbled.

1. Introduction

A method has been developed for specifying the characteristics of a real-time distributed application, using Mascot as the underlying framework. That method is described in Chapter 3 of the report **Distributed System Design Using Mascot and m-Verdi** [2].

Mascot is a design method supported by development tools and a run-time kernel [1]. The key concept in the approach is the modularization of the application into a Mascot network. In a Mascot network, activities are processing modules, which run concurrently. Channels and pools provide the data interfaces between activities. Mascot networks may also include devices, with servers providing the interface between activities and devices. Data components provide data access facilities at windows, and active components request access facilities at ports. A port is connected by a path to the window which provides the required access.

Synchronisation in Mascot covers the mutual exclusion of competing processes and the cross-stimulation of co-operating processes. Synchronisation is achieved by primitives which operate on special objects called control queues. Since synchronisation takes place only in respect to IDA's and servers, each control queue is conceptually part of the structure of an IDA or server. Synchronisation primitives are only available within an access procedure provided by an IDA or server.

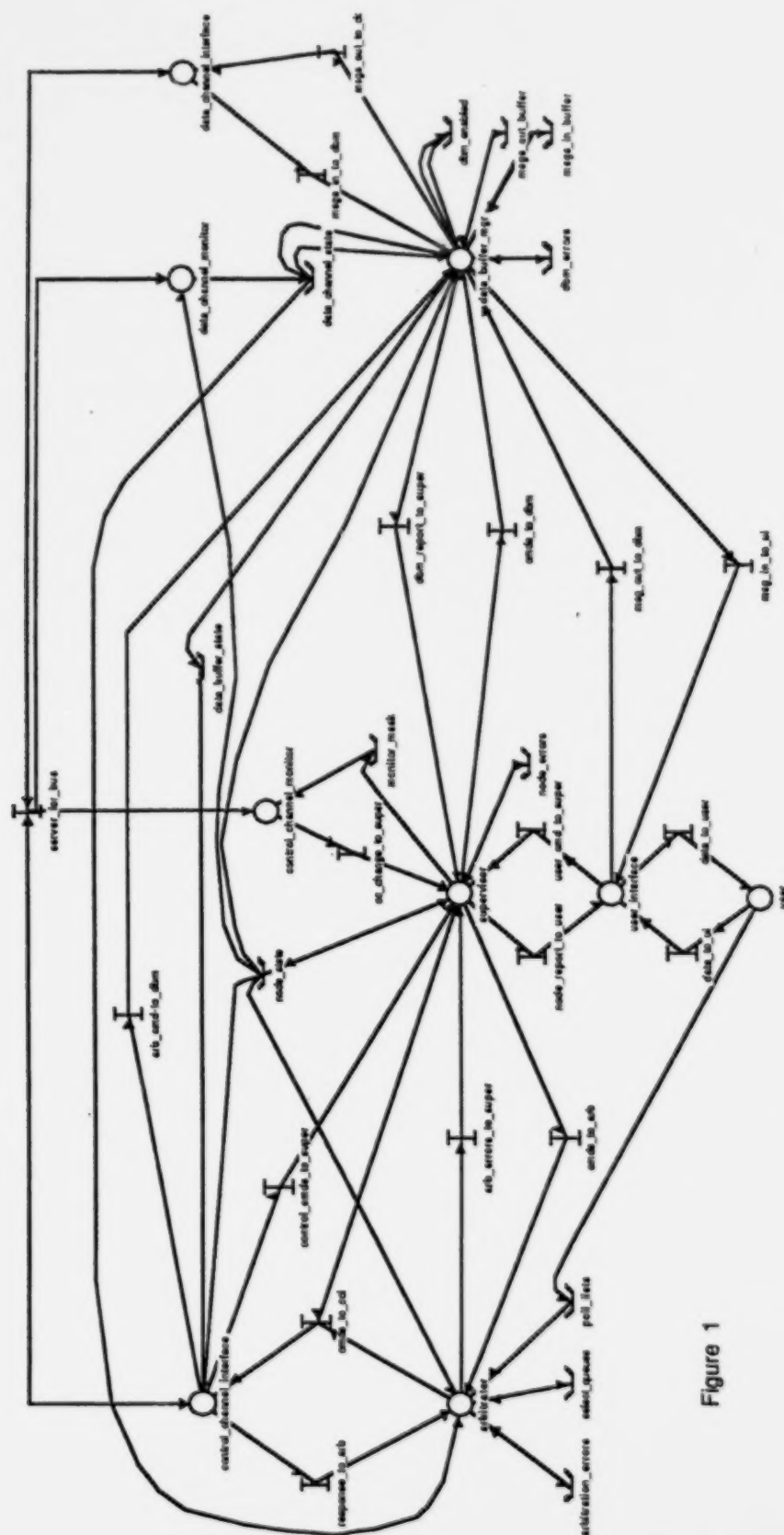
The guidelines [2, Chapter 3] outline the design method, which allows procedural specifications to be written for the activity and data components of a Mascot network. The Mascot model used is a modification of Mascot Version 3 [1] which is described in Chapter 2 of [2]. The abstract procedural statements are intended to express the real-time constraints on individual system components and the relationships of activities to data components.

This procedural specification method was used to specify parts of the SHINPADS [3] system.

First, a Mascot network diagram for an entire SHINPADS node was developed. The skeletal form of this network diagram is shown in Figure 1. (See Chapter 2 of [2] for a description of full v.s. skeletal network diagrams.) Procedural specifications were then written in m-Verdi [4] for the arbitration activity and for the IDA by which commands are sent from the supervisor to the arbitrator. Those specifications follow in Sections 2 and 3. Figures 2 and 3 show the Mascot network components which are used in each specification, with their names and types.

Finally, procedural specifications were also written for a bus channel connected to two servers. This specification is a simplification of the SHINPADS bus, which has six channels connected to many servers. Two separate bus specifications are found in Section 3. The high-level specification describes the requirements of the bus with respect to the windows of the servers attached to the bus; these windows provide users with access to the bus. The low-level specification describes the requirements with respect to the ports of the servers which are attached to the bus. Figures 4 and 5 show the Mascot network components used in these two levels of specification.

The SHINPADS node descriptions were studied carefully in an effort to make the network design and the specifications as accurate as possible. However, since many details of the SHINPADS system are imbedded deep in the firmware, the specifications may not be accurate in



SHINPADS Specifications

every detail. Furthermore, the SHINPADS implementation sometimes employs optimizations which add to the efficiency of the system, but which do not conform with the one's conceptual understanding of the system. In such cases, the network design developed attempts to be conceptually clear rather than technically accurate.

SHINPADS Specifications

2. Specification of Arbitrator Activity

{ m-Verdi specification for the arbitration activity of SHINPADS node }

program arbitrator =

environment

{ include declarations }

include "decls"

{ include general-purpose utility functions }

include "util"

{ include individual goodness functions }

include "okarbmsg"

include "okchoose"

include "okclrerr"

include "okdatach"

include "okerr"

include "okerrget"

include "oknodest"

include "okpolans"

include "okqadd"

include "okqhead"

include "okqrem"

include "okreset"

include "oksender"

include "oksuper"

{ include top-level goodness function }

include "correct"

{ include channel and pool access procedures }

include "chnfmsup"

include "chntosup"

include "chnfmcci"

include "chntocci"

include "nodepool"

include "chanpool"

include "qpools"

include "errpool"

include "pollpool"

{ abstract specification of the main program. Note that this must be in the environment section, since every definition outside of the environment must be provided with a model }

```
prog procedure arbitrator_stub(mvar _external_spec,
    mvar from_super_port, mvar to_super_port,
    mvar from_cci_port, mvar to_cci_port,
    mvar node_state_port, mvar data_channel_state_port,
    mvar select_queues_port, mvar arb_errors_port,
    mvar poll_lists_port) =
```

pre

_is_empty(_history(_external_spec, myself))

post

good(_external_spec)

end environment

```
main prog procedure arbitrator_spec(mvar _external_spec,
    mvar from_super_port, mvar to_super_port,
    mvar from_cci_port, mvar to_cci_port,
    mvar node_state_port, mvar data_channel_state_port,
    mvar select_queues_port, mvar arb_errors_port,
    mvar poll_lists_port) =
pre
    _is_empty(_history(_external_spec, myself))
post
    good(_external_spec)
begin
    arbitrator_stub(_external_spec,
        from_super_port, to_super_port,
        from_cci_port, to_cci_port,
        node_state_port, data_channel_state_port,
        select_queues_port, arb_errors_port, poll_lists_port)
end arbitrator_spec
```

end arbitrator

DECLS

June 24, 1988 13:22

Page 1

```

{ declarations of types, variables and functions used in rest of specification }

var i, j, k, m: int

{ channel messages }

prog type _msg_command_type = enum(RESET, ERROR_REQUEST)
  prog var a_command_kind: _msg_command_type

prog type _msg_kind_type =
  enum(POLL_REQUEST, SELECT, TERMINATE, REQUEST_DENIED)
  prog var a_msg_kind: _msg_kind_type

prog type _priority_type = enum(PRIORITY_0, PRIORITY_1, PRIORITY_2, PRIORITY_3)
  prog var a_priority: _priority_type
  prog var a_queue: _priority_type

prog type _response_type = enum(HAS_MESSAGE, HAS_NO_MESSAGE)
  prog var a_response: _response_type

prog type _err_report_type { format of report unspecified }
  prog var an_err_report: _err_report_type

{ data associated with 'put' event for channel of type arb-cmd }
prog type _arb_cmd_put_type = record
  field dest_node: int,
  field msg_kind: _msg_kind_type
end _arb_cmd_put_type

var an_arb_cmd: _arb_cmd_put_type
function _arb_cmd_put_dest_node_id(an_arb_cmd): int
function _arb_cmd_put_msg_kind(an_arb_cmd): _msg_kind_type

{ data associated with 'get' event for channel of type node-response }
prog type _node_response_get_type = record
  field source_node_id: int,
  field node_response: _response_type,
  field priority: _priority_type
end _node_response_get_type

var a_node_response: _node_response_get_type
function _node_response_get_source_node_id(a_node_response): int
function _node_response_get_response_kind(a_node_response):
  _response_type
function _node_response_get_priority(a_node_response): _priority_type

{ data associated with 'write' event for channel of type error-report }
prog type _error_report_write_type = record
  field error_report: _err_report_type
end _error_report_write_type

var an_error_report: _error_report_write_type
function _error_report_write_report(an_error_report): _err_report_type

```

```

{ data associated with 'get' event for channel of type super-cmd }
prog type _super_cmd_get_type = record
    field command_kind: _msg_command_type
end _super_cmd_get_type

var a_super_cmd: _super_cmd_get_type
function _super_cmd_get_command_kind(a_super_cmd): _super_cmd_get_type

{ pool data }

prog type _err_id_type = enum(POLL_TIMEOUT_ERR, QUEUE_OVERFLOW_ERR,
    NODE_FAILED_TO_SEND, NODE_MSG_TOO_LONG)
prog var an_err_id: _err_id_type

prog type _node_state_type = enum(IS_BUS_CONTROLLER, IS_NOT_BUS_CONTROLLER)
prog var a_node_state: _node_state_type

prog type _node_list_type = enum(PRIMARY, SECONDARY)
prog var a_node_list: _node_list_type

prog type _node_id_type = those i: int'ge(i,0)
prog var a_node_id: _node_id_type
prog var a_dest_node_id: _node_id_type
prog var a_source_node_id: _node_id_type

prog type _data_channel_state_type = enum(BUSY, IDLE)
prog var a_data_channel_state: _data_channel_state_type

prog type _data_channel_msg_id_type = enum(MESSAGE_A, MESSAGE_B, NIL_MESSAGE)
prog var a_data_channel_msg_id: _data_channel_msg_id_type

prog type _sequence_of_node_ids_type
var a_primary_list: _sequence_of_node_ids_type
var a_secondary_list: _sequence_of_node_ids_type
var a_node_id_sequence: _sequence_of_node_ids_type
function list_is_empty(a_node_id_sequence): bool
function list_apr(a_node_id_sequence, a_node_id): _sequence_of_node_ids_type
function list_length(a_node_id_sequence): int
function list_right(a_node_id_sequence): _node_id_type

function list_item(a_node_id_sequence, i): _node_id_type

prog var a_subcycle_count: int

{ data associated with 'get' event for pool of type state-of-node }
prog type _state_of_node_get_type = record
    field state: _node_state_type
end state_of_node_get_type
var some_node_state_data: _state_of_node_get_type
function _state_of_node_get_state(some_node_state_data): _node_state_type

{ data associated with 'get' event for pool of type state-of-data-channel }

```

```

prog type _state_of_data_channel_get_type = record
    field state: _data_channel_state_type,
    field msg_id: _data_channel_msg_id_type
end _state_of_data_channel_get_type

var some_channel_state_data: _state_of_data_channel_get_type
function _state_of_data_channel_get_state(some_channel_state_data):
    _data_channel_state_type
function _state_of_data_channel_get_msg_id(some_channel_state_data):
    _data_channel_msg_id_type

{ data associated with 'get-next-to-poll' event for pool of type lists }
prog type _lists_get_next_to_poll_type = record
    field node_list: _node_list_type,
    field node_index: int,
    field subcycle_count: int,
    field primary_list: _sequence_of_node_ids_type,
    field secondary_list:
        _sequence_of_node_ids_type
end _lists_get_next_to_poll_type

var some_poll_list_data: _lists_get_next_to_poll_type
function _lists_get_next_to_poll_node_list(some_poll_list_data):
    _node_list_type
function _lists_get_next_to_poll_node_index(some_poll_list_data): int
function _lists_get_next_to_poll_subcycle_count(some_poll_list_data): int
function _lists_get_next_to_poll_primary_list(some_poll_list_data):
    _sequence_of_node_ids_type
function _lists_get_next_to_poll_secondary_list(some_poll_list_data):
    _sequence_of_node_ids_type

function _lists_get_next_to_poll_node_ident(some_poll_list_data): int =
{ node id in primary or secondary list at position
  _list_get_next_to_poll_node_index;
  _lists_get_next_to_poll_node_list determines which list to look in }
begin
    if
        _lists_get_next_to_poll_node_list(some_poll_list_data) =
            _node_list_type'PRIMARY
    then
        list_item( _lists_get_next_to_poll_primary_list(
            some_poll_list_data),
            _lists_get_next_to_poll_node_index(some_poll_list_data))
    else
        list_item( _lists_get_next_to_poll_secondary_list(
            some_poll_list_data),
            _lists_get_next_to_poll_node_index(some_poll_list_data))
    end if
end _lists_get_next_to_poll_node_ident

{ data associated with 'get-head, remove-node, or put-node' event for pool
  of type queues }
{ Note: this data type is used for event data from three different access
  procedures, get_head, remove_node, and put_node;
  the name 'get_or_put' is used instead }
prog type _queues_get_or_put_type = record
    field node_id: int,

```

```

        field queue: _priority_type
    end _queues_get_or_put_type

    var some_queue_data: _queues_get_or_put_data
    function _queues_get_or_put_node_id(some_queue_data): int
    function _queues_get_or_put_queue(some_queue_data): _priority_type

    { data associated with 'write' event for pool of type errors }
    prog type _errors_write_type = record
        field err_id: _err_id_type,
        field node_id: int,
        field queue: _priority_type
    end errors_write_type

    var some_write_error_data: _errors_write_type
    function _errors_write_err_id(some_write_error_data): _err_id_type
    function _errors_write_node_id(some_write_error_data): int
    function _errors_write_queue(some_write_error_data): _priority_type

    { data associated with 'read' event for pool of type errors }
    prog type _errors_read_type = record
        field err_report: _err_report_type
    end errors_read_type

    var some_read_error_data: _errors_read_type
    function _errors_read_err_report(some_read_error_data): _err_report_type

    { id's of components }

    type _ID_type = those int'dummy: true
    var _an_ID: _ID_type
    { ID's of the IDA's used in this specification }
    type ida_id = enum(
        FROM_SUPER_ID,
        TO_SUPER_ID,
        ERRORS_ID,
        NODE_STATE_ID,
        DATA_CHANNEL_STATE_ID,
        POLL_LISTS_ID,
        SELECT_QUEUES_ID,
        TO_CCI_ID,
        FROM_CCI_ID
    )

    { ID of the activity used in this specification,
      namely the arbitrator itself }
    prog const myself: _ID_type

    function the_ID(ida_id'dummy): _ID_type
    { maps ID onto unique integer }
    axiom the_ID_is_one_to_one_axiom(ida_id'dummy) =
    begin
        all ida_id'dummy,
        ida_id'dummy'0:
            implies
            (
                the_ID(ida_id'dummy) = the_ID(ida_id'dummy'0),
                ida_id'dummy = ida_id'dummy'0
            )
    end

```

```

    )
    end the_ID_is_one_to_one_axiom

```

```

{ events }

```

```

type _event_return = enum(NO_MESSAGE, TIMED_OUT, POOL_FULL, POOL_EMPTY, OK)

```

```

type _event_kind = enum(
    NIL_EVENT,
    _SUPER_CMD_GET_KIND,
    _ERROR_REPORT_WRITE_KIND,
    _ERRORS_CLEAR_KIND,
    _ERRORS_WRITE_KIND,
    _ERRORS_READ_KIND,
    _STATE_OF_NODE_GET_KIND,
    _STATE_OF_DATA_CHANNEL_GET_KIND,
    _LISTS_RESET_KIND,
    _LISTS_GET_NEXT_TO_POLL_KIND,
    _QUEUES_GET_HEAD_KIND,
    _QUEUES_PUT_NODE_KIND,
    _QUEUES_REMOVE_NODE_KIND,
    _ARB_CMD_PUT_KIND,
    _POLL_RESPONSE_GET_KIND
)

```

```

type _event_type

```

```

    var _event: _event_type
    function _kind(_event): _event_kind
    function _time(_event): int
    function _return_value(_event): _event_return
    function _passive_ID(_event): _ID_type
    function _active_ID(_event): _ID_type

    function _arb_cmd_put_extract(_event): _arb_cmd_put_type
    function _node_response_get_extract(_event): _node_response_get_type
    function _error_report_write_extract(_event): _error_report_write_type
    function _super_cmd_get_extract(_event): _super_cmd_get_type

    function _state_of_node_get_extract(_event): _state_of_node_get_type
    function _state_of_data_channel_get_extract(_event):
        _state_of_data_channel_get_type
    function _lists_get_next_to_poll_extract(_event):
        _lists_get_next_to_poll_type
    function _queues_get_or_put_extract(_event): _queues_get_or_put_type
    function _errors_write_extract(_event): _errors_write_type
    function _errors_read_extract(_event): _errors_read_type

    { need these two functions for TO_CCI event with msg kind SELECT only }
    { msg-id and priority are stored in event but not sent to CCI }
    function _select_message_id(_event): _data_channel_msg_id_type
    function _select_priority(_event): _priority_type

```

```

{ event sequences }

```

```

type _event_sequence_type

```

```

    var _an_event-sequence: _event_sequence_type

```

```

var _an_event_kind: _event_kind

{ functions which define a sequence }
function _is_empty(_an_event_sequence): bool
function _apr(_an_event_sequence, _event): _event_sequence_type
function _length(_an_event_sequence): int
function _right(_an_event_sequence): _event

{ sequence functions which may be specified in terms of previous functions }
function _dropr(_an_event_sequence): _event_sequence_type
function _last_event(_an_event_sequence, _an_event_kind, _an_ID):
    _event_type
    { returns the last event of given kind on given IDA }
    { returns event of type NIL_EVENT if none }
function _item(_an_event_sequence, i): _event_type
function _inrange(_an_event_sequence, i): bool
function _first_i_events(_an_event_sequence, i): _event_sequence_type
function _left(_an_event_sequence): _event_type

var _hist : _event_sequence_type
const empty_sequence: _event_sequence_type

{ all events that have occurred in the system }
prog type IDA_interface_spec
prog var _external_spec: IDA_interface_spec
    function _history(_external_spec, _an_ID): _event_sequence_type
    function _current_time(_external_spec): int
    function _well_sequenced(_external_spec): bool
    { each event on a particular component has a distinct time }

prog type _socket
    var _a_socket: _socket
    function _socket_passive_ID(_a_socket): _ida_id
    function _valid_active_ID(_a_socket, _an_ID): bool

{ error return used by access procedures - network specific }
prog var _err: int

{ ports of arbitrator activity }
prog var from_super_port: _socket
prog var to_super_port: _socket
prog var from_cci_port: _socket
prog var to_cci_port: _socket
prog var node_state_port: _socket
prog var data_channel_state_port: _socket
prog var select_queues_port: _socket
prog var arb_errors_port: _socket
prog var poll_lists_port: _socket

```

{ general purpose functions used by the goodness functions }

```
function is_arbitration_event(_event): bool =
  { returns true if given event can take place only when node is bus
    controller }
  { omits supervisor commands, node state checks, errors }
  { cannot actually write errors unless arbitrator, but do not include
    errors here }
  begin
    or (
      _kind(_event) = _event_kind' _LISTS_GET_NEXT_TO_POLL_KIND,
      _kind(_event) = _event_kind' _QUEUES_GET_HEAD_KIND,
      _kind(_event) = _event_kind' _QUEUES_PUT_NODE_KIND,
      _kind(_event) = _event_kind' _ARB_CMD_PUT_KIND,
      _kind(_event) = _event_kind' _POLL_RESPONSE_GET_KIND,
      _kind(_event) = _event_kind' _STATE_OF_DATA_CHANNEL_GET_KIND
    )
  end is_arbitration_event
```

```
function event_is_bus_controller(_event): bool =
  { returns true if event is read of node status, and node is bus controller }
  begin
    and (
      _kind(_event) = _event_kind' _STATE_OF_NODE_GET_KIND,
      _passive_ID(_event) = the_ID(ida_id' _NODE_STATE_ID),
      _state_of_node_get_type(_state_of_node_get_extract(_event)) =
        _node_state_type' IS_BUS_CONTROLLER
    )
  end event_is_bus_controller
```

```
function arb_event_valid(_hist, _event): bool =
  { returns true if node is in bus controller mode and therefore the
    arbitration event can occur }
  pre
    is_arbitration_event(_event)
  begin
    if _length(_hist) = 0
    then false
    else
      event_is_bus_controller(_last_event(_hist,
        _event_kind' _STATE_OF_NODE_GET_KIND,
        the_ID(ida_id' _NODE_STATE_ID)))
    end if
  end arb_event_valid
```

```
function starting_bus_controller_mode(_hist, _event): bool =
  { returns true if event is read of node status, and node is bus controller,
    and node was not bus controller prior to this event i.e. the node is
    just starting a cycle as the bus controller }
  begin
    and (
      event_is_bus_controller(_event),
      { is now bus controller }
      if _length(_hist) = 0
      then true
      else

```

```

        event_is_bus_controller(_last_event(_hist,
            _event_kind' STATE_OF_NODE_GET_KIND,
            theID(ida_id'NODE_STATE_ID))) = false
            { was not bus controller when last checked }
    end if
)
end starting_bus_controller_mode

function arb_events(_hist): _event_sequence_type =
    { get sequence of all events since last became bus controller, omitting
      status checks, supervisor commands, and error handling }
    measure ord_val(_length(_hist))
begin
    if _is_empty(_hist)
    then empty_sequence
    else
        if
            and (
                _kind(_right(_hist)) =
                    _event_kind' STATE_OF_NODE_GET_KIND,
                _passive_ID(_right(_hist)) = the_ID(ida_id'NODE_STATE_ID),
                _state_of_node_get_state(_state_of_node_get_extract(
                    _right(_hist))) =
                    _node_state_type' IS_NOT_BUS_CONTROLLER
            )
        then empty_sequence
        else
            if is_arbitration_event(_right(_hist))
            then
                _apr(arb_events(_dropr(_hist), _right(_hist)))
                { take this event - it is an arbitration event }
            else
                arb_events(_dropr(_hist))
                { omit this event - it is not arbitration event }
            end if
        end if
    end if
end if
end arb_events

const the_nil_event: _event_type
axiom
    all _event:
        implies
            (
                _kind(_event) = _event_kind'NIL_EVENT,
                _event = the_nil_event
            )

function next_arb_event(_hist, i): _event =
    { find the next arbitration event which follows the i'th event and which is
      in the same arbitration phase }
    measure ord_val(minus(_length(_hist), i))
begin
    if
        int'le(_length(_hist), i)
    then
        the_nil_event
    end if
end

```

```

    ( quit when no i+1'th event )
else
  if
    is_arbitration_event(_item(_hist, plus(i,1)))
  then
    _item(_hist, plus(i,1))
    { i+1'th event is arbitration event - return it }
  else
    if { stop controlling bus }
      and (
        _kind(_item(_hist, plus(i,1))) =
          _event_kind' STATE_OF_NODE_GET_KIND,
        _passive_ID(_item(_hist, plus(i,1))) =
          the_ID(ida_id'NODE_STATE_ID),
        _state_of_node_get_state(_state_of_node_get_extract(
          _item(_hist, plus(i,1)))) =
          _node_state_type' IS_NOT_BUS_CONTROLLER
      )
    then the_nil_event
    else
      next_arb_event(_hist, plus(i, 1))
      { keep looking ahead in event sequence }
    end if
  end if
end if
end next_arb_event

```

```

var a_msg_kind: _msg_kind_type

```

```

function last_cci_msg_sent(_hist, a_msg_kind): _event_type =
  measure ord_val(_length(_hist))
begin
  { returns most recent event in history which has kind ARB_CMD_PUT_KIND
    and which has the required kind of message }
  if _is_empty(_hist)
    then the_nil_event
  else
    if
      and (
        _kind(_right(_hist)) = _event_kind' ARB_CMD_PUT_KIND,
        _passive_ID(_right(_hist)) = the_ID(ida_id'TO_CCI_ID),
        _arb_cmd_put_msg_kind(_arb_cmd_put_extract(
          _right(_hist))) =
          a_msg_kind
      )
    then
      _right(_hist)      { this is event we want }
    else
      if
        and (
          _kind(_right(_hist)) =
            _event_kind' STATE_OF_NODE_GET_KIND,
          not(event_is_bus_controller(_right(_hist))),
        )
      then
        the_nil_event
      else

```

```

        { stop is no longer bus controller }
    else
        last_cci_msg_sent(_dropr(_hist))
        { keep looking farther back in sequence }
    end if
end if
end if
end last_cci_msg_sent

```

```
var msg_id: _data_channel_msg_id_type
```

```

function complement(msg_id): _data_channel_msg_id_type =
{ the data channel state event returns a message id which
initially is NIL_MESSAGE and then changes to MESSAGE_A when the first
message has started to be transmitted on the data channel;
it then switches between MESSAGE_A and MESSAGE_B whenever another message
start has been detected }
begin
    if
        or (
            msg_id = _data_channel_msg_id_type'NIL_MESSAGE,
            msg_id = _data_channel_msg_id_type'MESSAGE_B
        )
    then
        _data_channel_msg_id_type'MESSAGE_A
    else
        _data_channel_msg_id_type'MESSAGE_B
    end if
end complement

```

```
var time_have_waited: int
```

```

function length_of_wait(_hist, _event, time_have_waited): int =
{ given data_channel_state which indicates that channel is idle,
decide how long have been waiting with idle channel after sending select
for msg; time is actually time from first idle-state event after the
select for the msg to this idle-state event }
pre
    and (
        _kind(_event) = _event_kind'_STATE_OF_DATA_CHANNEL_GET_KIND,
        _passive_ID(_event) = the_ID(ida_id'DATA_CHANNEL_STATE_ID)
    )
measure ord'val(_length(_hist))
begin
    if
        or (
            _length(_hist) = 0,
            _state_of_data_channel_get_state(
                _state_of_data_channel_get_extract(_event))
            <> _data_channel_state_type'IDLE
        )
    then 0
    else
        if
            and (
                _kind(_right(_hist)) = _event_kind'_ARB_CMD_PUT_KIND,

```

```

    _passive_ID(_right(_hist)) = the_ID(ida_id'TO_CCI_ID),
    _arb_cmd_put_msg_kind(_arb_cmd_put_extract(
        _right(_hist))) =
        _msg_kind_type'SELECT
    { before called this function have already checked that
      this select is one for msg which has not started to send }
  )
then
  time_have_waited
  { stop when get back to select for the message }
else
  if
    and (
      _kind(_right(_hist)) =
        _STATE_OF_DATA_CHANNEL_GET_KIND,
      _passive_ID(_right(_hist)) =
        the_ID(ida_id'DATA_CHANNEL_STATE_ID),
      _state_of_data_channel_get_state(
        _state_of_data_channel_get_extract(
          _right(_hist))) = _data_channel_state_type'IDLE,
      _state_of_data_channel_get_msg_id(
        _state_of_data_channel_get_extract(
          _right(_hist))) =
        _state_of_data_channel_get_msg_id(
          _state_of_data_channel_get_extract(_event))
      { was waiting for same msg to start }
    )
  then
    length_of_wait(_dropr(_hist), _event,
      minus(_time(_event), _time(_right(_hist)))
      { change wait length to time from this event to
        given event and continue looking farther back in
        sequence }
    )
  else
    length_of_wait(_dropr(_hist), _event, time_have_waited)
    { keep looking farther back in sequence }
  end if
end if
end if
end length_of_wait

```

```
const MAXIMUM_SEND_DELAY: int
```

```

function node_failed_to_send(_hist, _event): bool =
  { returns true if msg selected, but has not started to send after
    MAXIMUM_SEND_DELAY }
pre
  and (
    _kind(_event) = _event_kind'_STATE_OF_DATA_CHANNEL_GET_KIND,
    _passive_ID(_event) = the_ID(ida_id'DATA_CHANNEL_STATE_ID)
  )
begin
  if
    and (
      _state_of_data_channel_get_state(
        _state_of_data_channel_get_extract(_event))
        = _data_channel_state_type'IDLE,
    )
  then
    length_of_wait(_dropr(_hist), _event, time_have_waited)
    { keep looking farther back in sequence }
  end if
end if
end length_of_wait

```

```

    _state_of_data_channel_get_msg_id(
        _state_of_data_channel_get_extract(_event))
    = _select_message_id(last_cci_msg_sent(_hist,
        _msg_kind_type'SELECT))
    { last msg selected has not started to send and channel is
      idle }
    { Note: msgid in SELECT event is id of previous msg sent;
      therefore if select msgid is MSG_A, state will be MSG_B
      while it is sending (or sent) }
  )
then
  int'gt(
    length_of_wait(_hist, _event, 0), MAXIMUM_SEND_DELAY
  )
else
  false
end node_failed_to_send

function time_for_select(_event, _hist): bool =
  { if this is first message trying to send in arbitration cyle, then
    data channel should be idle, and has not started to send any message;
    otherwise, data channel state must indicate that that previous
    message has started to be sent (or finished) }
  { Note: if a msg previously selected for sending is not sent, then after a
    sufficiently long wait, another select is permitted }
  { hist is arbitration events in same arbitration phase }
begin
  and (
    _kind(_event) = _event_kind'STATE_OF_DATA_CHANNEL_GET_KIND,
    _passive_ID(_event) = the_ID(ida_id'DATA_CHANNEL_STATE_ID),
    or (
      and (
        { first select in arbitration cycle - assume initial state
          of data channel is IDLE and NIL_MESSAGE }
        _kind(last_cci_msg_sent(_hist, _msg_kind_type'SELECT)) =
          _event_kind'NIL_EVENT,
        _state_of_data_channel_get_state(
          _state_of_data_channel_get_extract(
            _event)) =
          _data_channel_state_type'IDLE,
        _state_of_data_channel_get_msg_id(
          _state_of_data_channel_get_extract(
            _event)) =
          _data_channel_msg_id_type'NIL_MESSAGE
      ),
      and (
        { last msg select has started to send }
        _kind(last_cci_msg_sent(_hist, _msg_kind_type'SELECT)) =
          _event_kind'NIL_EVENT,
        _state_of_data_channel_get_msg_id(
          _state_of_data_channel_get_extract(_event)) =
          complement(_select_message_id(
            last_cci_msg_sent(_hist,
              _msg_kind_type'SELECT)))
      end if
    ),
    and (
      { msg selected for sending has not been sent }

```

```

        _kind(last_cci_msg_sent(_hist, _msg_kind_type'SELECT)) <>
        _event_kind'NIL_EVENT,
        node_failed_to_send(_hist, _event)
    )
)
end time_for_select

function length_of_send(_hist, _event, time_have_waited): int =
{ given data_channel_state which indicates that channel is busy,
  decide how long have been waiting with busy channel since selected that
  msg; time is actually time from first busy-state event after the select
  to this busy-state event }
pre
    and (
        _kind(_event) = _event_kind' STATE_OF_DATA_CHANNEL_GET_KIND,
        _passive_ID(_event) = the_ID(ida_id'DATA_CHANNEL_STATE_ID)
    )
measure ord'val(_length(_hist))
begin
    if
        or (
            _length(_hist) = 0,
            (_state_of_data_channel_get_extract(_event))
            <> _data_channel_state_type'BUSY
        )
    then 0
    else
        if
            or (
                { select for the message which is still sending }
                and (
                    _kind(_right(_hist)) = _event_kind' ARB_CMD_PUT_KIND,
                    _passive_ID(_right(_hist)) = the_ID(ida_id'TO_CCI_ID),
                    _arb_cmd_put_msg_kind(_arb_cmd_put_extract(
                        _right(_hist)))
                    = _msg_kind_type'SELECT,
                    _state_of_data_channel_get_msg_id(
                        _state_data_channel_get_extract(_event)) =
                    complement(_state_of_data_channel_get_msg_id(
                        _state_of_data_channel_get_extract(
                            _right(_hist))))
                    { Note: msgid in SELECT event is previous msg sent
                      or sending; therefore if busy with MSG_A, select
                      for it has MSG_B and vice verse }
                ),
                { data channel was idle at this point }
                and (
                    _kind(_right(_hist)) =
                        STATE_OF_DATA_CHANNEL_GET_KIND,
                    _passive_ID(_right(_hist)) =
                        the_ID(ida_id'DATA_CHANNEL_STATE_ID),
                    _state_of_data_channel_get_state(
                        _state_of_data_channel_get_extract(
                            _right(_hist))) = _data_channel_state_type'IDLE
                )
            )
        )
    end if
end if
end function

```

```

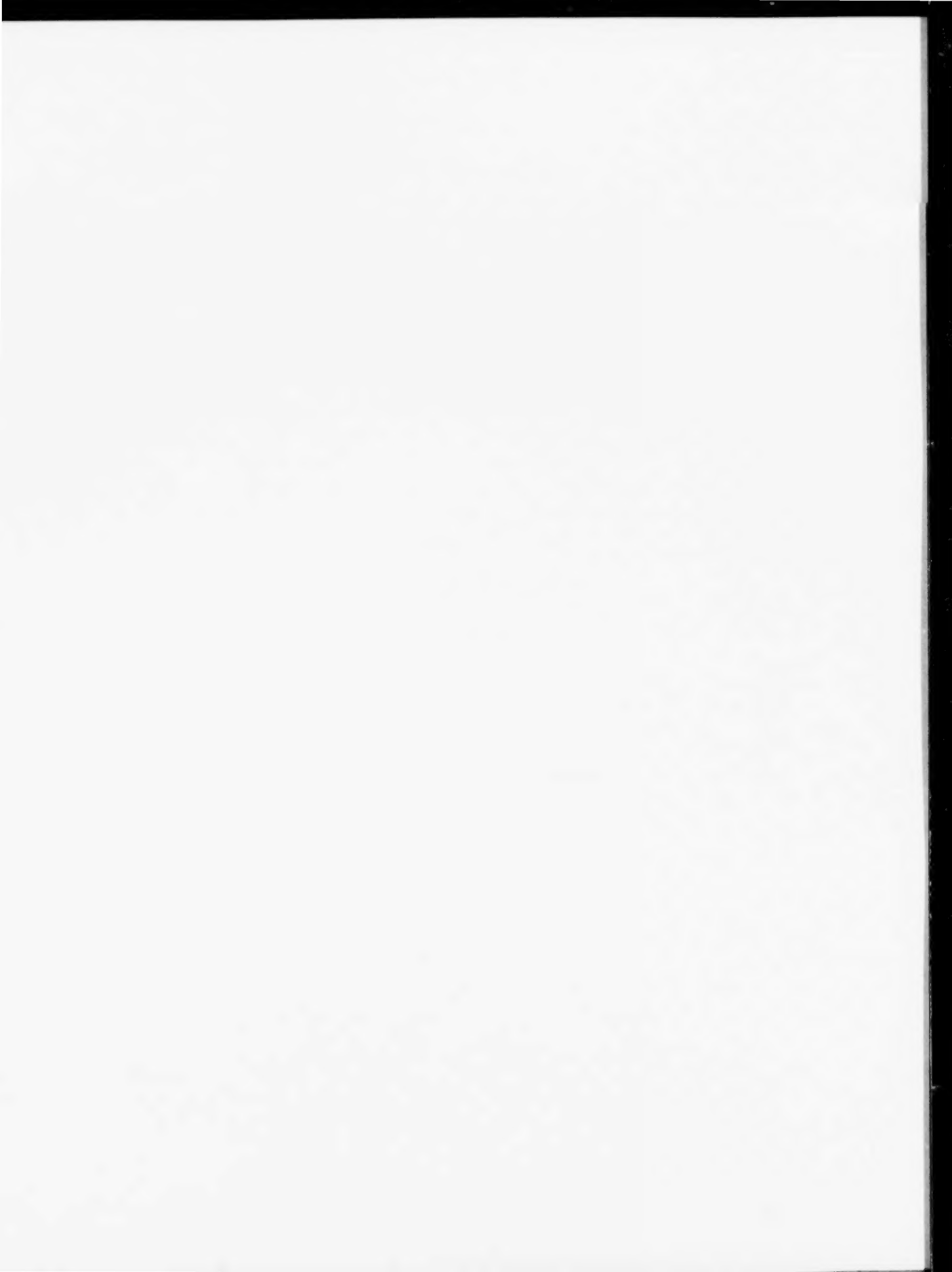
then
    time_have_waited
    { stop when get select for msg which is still sending, or
      when get back to channel idle state }
else
    if
        and (
            _kind(_right(_hist)) =
                _STATE_OF_DATA_CHANNEL_GET_KIND,
            _passive_ID(_right(_hist)) =
                the_ID(ida_id'DATA_CHANNEL_STATE_ID),
            _state_of_data_channel_get_state(
                _state_of_data_channel_get_extract(
                    _right(_hist))) = _data_channel_state_type'BUSY,
            _state_of_data_channel_get_msg_id(
                _state_of_data_channel_get_extract(_right(_hist)))
                = _state_of_data_channel_get_msg_id(
                    _state_of_data_channel_get_extract(_event))
            { was waiting for same msg to complete }
        )
    then
        length_of_wait(_dropr(_hist), _event,
            minus(_time(_event), _time(_right(_hist))))
        { change send length to time from this event to
          given event and continue looking farther back in
          sequence }
    else
        length_of_wait(_dropr(_hist), _event, time_have_waited)
        { continue looking farther back in sequence }
    end if
end if
end if
end length_of_send

const MAXIMUM_SEND_TIME: int

function node_sent_too_long(_hist, _event): bool =
    pre
        and (
            _kind(_event) = _event_kind'_STATE_OF_DATA_CHANNEL_GET_KIND,
            _passive_ID(_event) = the_ID(ida_id'DATA_CHANNEL_STATE_ID)
        )
    begin
        if
            and (
                data_channel_state(_state_of_data_channel_get_event(_event)) =
                    _data_channel_state_type'BUSY,
                _state_of_data_channel_get_msg_id(
                    _state_of_data_channel_get_extract(_event))
                    = _select_message_id(last_cci_msg_sent(_hist,
                        _msg_kind_type'SELECT)),
                { last msg selected has not started to send and still
                  sending previous one }
            )
        then
            int'gt(
                length_of_send(_hist, _event, 0), MAXIMUM_SEND_TIME
            )
        end if
    end
end function

```

```
        )  
    else  
        false  
end node_sent_too_long
```



```

{ functions to verify goodnes with respect to events of kind
  put-arbitration-command (to cci) }

{ arbitration commands are of four kinds - SELECT, TERMINATE,
  REQUEST_DENIED, or POLL_REQUEST; goodness with respect to each kind of
  command is checked separately }

function select_seq_ok(_hist): bool =
begin
  { queue_select_head_seq_ok says that after get head, send select }
  { so do not have to say it here }

  { save the message id from the last data channel state check in the
    select event }
  { save the priority from the last get-head-of-queue in the select event}
  all i:
    implies
      (
        and (
          _inrange(_hist, i),
          _kind(_item(_hist, i)) = _event_kind'ARB_CMD_PUT_KIND,
          _passive_ID(_item(_hist, i)) = the_ID(ida_id'TO_CCI_ID),
          _arb_cmd_put_msg_kind(_arb_cmd_put_extract(
            _item(_hist, i)))
            = _msg_kind_type'SELECT
          ),
          and (
            _kind(_last_event(_first_i_events(_hist, minus(i,1)),
              _event_kind'STATE_OF_NODE_GET_KIND,
              the_ID(ida_id'NODE_STATE_ID))) <>
              _event_kind'NIL_EVENT,
            _state_of_data_channel_get_msg_id(
              _state_of_data_channel_get_extract(_last_event(
                _first_i_events(_hist, minus(i,1)),
                _event_kind'STATE_OF_NODE_GET_KIND,
                the_ID(ida_id'NODE_STATE_ID))))
              = _select_message_id(_item(_hist, i)),
            _queues_get_or_put_queue(_queues_get_or_put_extract(
              _last_event(_first_i_events(_hist, minus(i,1)),
              _event_kind'QUEUES_GET_HEAD_KIND,
              the_ID(ida_id'SELECT_QUEUES_ID))))
              = _select_priority(_item(_hist, i))
          )
        )
      )
end select_seq_ok

function request_deny_seq_ok(_hist): bool =
begin
  { have already said in ok_err that queue full follwed by request denied}
end request_deny_seq_ok

function message_terminate_seq_ok(_hist): bool =
begin
  { if event is queueing of priority 0 message, and previous select
    was for message of greater than 0, send terminate of previous

```

```

message )
all i:
  implies
  (
    and (
      _inrange(_hist, i),
      _kind(_item(_hist, i)) =
        _event_kind' QUEUES_PUT_NODE_KIND,
      _passive_ID(_item(_hist, i)) =
        the_ID(ida_id' SELECT_QUEUES_ID),
      _return_value(_item(_hist, i)) = _event_return'OK,
      _queues_get_or_put_queue(_queues_get_or_put_extract(
        _item(_hist, i))) =
        _priority_type'PRIORITY_0,
      _kind(last_cci_msg_sent(arb_events(
        _first_i_events(_hist, minus(i,1))),
        _msg_kind_type'SELECT)) <>
        _event_return'NIL_EVENT
    ),
    if
      and (
        _kind(next_arb_event(_hist, i)) <> _event_kind'NIL,
        _select_priority(last_cci_msg_sent(
          arb_events(_first_i_events(_hist, minus(i,1))),
          _msg_kind_type'SELECT)) <>
          _priority_type'PRIORITY_0
      )
    then
      and (
        _kind(next_arb_event(_hist, i)) =
          _event_kind' ARB_CMD_PUT_KIND,
        _passive_ID(next_arb_event(_hist, i)) =
          the_ID(ida_id'TO_CCI_ID),
        _arb_cmd_put_msg_kind(_arb_cmd_put_extract(
          next_arb_event(_hist,i))) =
          _msg_kind_type'TERMINATE,
        _arb_cmd_put_dest_node_id(_arb_cmd_put_extract(
          next_arb_event(_hist, i))) =
          _arb_cmd_put_dest_node_id(_arb_cmd_put_extract(
            last_cci_msg_sent(
              arb_events(_first_i_events(_hist, minus(i,1))),
              _msg_kind_type'SELECT)))
        { terminate sent to node previously selected }
      )
    else
      true
    end if
  )
end message_terminate_seq_ok

```

```

function poll_node_seq_ok(_hist): bool =
  begin
    { after choose node to poll, send poll request }
    all i:
      implies
      (
        and (

```

```

    _inrange(_hist, i),
    _kind(_item(_hist, i)) = _event_kind'ARB_CMD_PUT_KIND,
    _passive_ID(_item(_hist, i)) = the_ID(ida_id'TO_CCI_ID),
    _arb_cmd_put_msg_kind(
        _arb_cmd_put_extract(_item(_hist, i))
        = _msg_kind_type'POLL_REQUEST
    ),
and (
    arb_events(_first_i_events(_hist, minus(i,1))) <>
        empty_sequence,
        { must be in midst of arbitration cycle }
    _kind(_right(arb_events(
        _first_i_events(_hist, minus(i,1))))
        = _event_kind'LISTS_GET_NEXT_TO_POLL_KIND,
    _passive_ID(_right(arb_events(_hist, minus(i,1))) =
        the_ID(ida_id'POLL_LISTS_ID),
    _return_value(_right(arb_events(_first_i_events(
        _hist, minus(i,1)))) =
        _event_return'OK,
    _lists_get_next_to_poll_node_ident(
        _lists_get_next_to_poll_extract(
            _right(arb_events(
                _first_i_events(_hist, minus(i,1))))))
        = _arb_cmd_put_dest_node_id(_arb_cmd_put_extract(
            _item(_hist, i)))
        { ask for response from same node which is next to poll}
    )
)
end poll_node_seq_ok

function send_arb_msg_append_event_ok(_hist, _event): bool =
begin
    if
        and (
            _kind(_event) = _event_kind'ARB_CMD_PUT_KIND,
            _passive_ID(_event) = the_ID(ida_id'TO_CCI_ID)
        )
    then
        and (
            arb_event_valid(_hist, _event),      { in bus controller mode }
            _return_value(_event) = _event'return'OK
        )
    else
        true
    end if
end send_arb_msg_append_event_ok

function ok_send_arb_msg(_hist): bool =
begin
    if _is_empty(_hist)
    then true
    else
        and (
            { have 4 kinds of msg; check sequence for each kind }
            select_seq_ok(_hist),

```

```
        request_deny_seq_ok(_hist),  
        message_terminate_seq_ok(_hist),  
        poll_node_seq_ok(_hist),  
        send_arb_msg_append_event_ok(_dropr(_hist), _right(_hist))  
    )  
end if  
end ok_send_arb_msg
```

```

{ functions used to verify goodness with respect to events of kind
  get-next-to-poll (from poll-lists pool) }

int: p, c, s
{ length of primary list, length of secondary list, subcycle count }

{ functions used in getting next node to poll from poll lists }
function secondary_nodes(_hist): _event_sequence_type =
  { selects get-next-node-to-poll events which returned secondary node }
  measure ord'val(_length(_hist))
  begin
    if _length(_hist) = 0
    then
      empty_sequence
    else
      if
        and (
          _kind(_right(_hist)) =
            _event_kind' LISTS_GET_NEXT_NODE_TO_POLL_KIND,
          _passive_ID(_right(_hist)) = the_ID(ida_id' POLL_LISTS_ID),
          _lists_get_next_to_poll_node_list(
            _lists_get_next_to_poll_extract(
              _right(_hist))) = SECONDARY
        )
      then
        _apr(_dropr(_hist), _right(_hist))
        { take event and keep looking }
      else
        secondary_nodes(_dropr(_hist))
        { omit event and keep looking }
      end if
    end if
  end secondary_nodes

function poll_cycle_check(_hist, p, s, c): bool =
  { checks that if have been polling for sufficiently long time, then
    have selected for polling all of the nodes in the primary list and
    the next c of the nodes in the secondary list }
  begin
    if
      and (
        int'ge(_length(_hist), plus(p,c)),
        { _length(_hist) >= p + c }

        all i:
          implies
            (
              _inrange(_hist,i),
              and (
                _kind(_item(_hist,i)) =
                  _event_kind' LISTS_GET_NEXT_TO_POLL_KIND,
                _passive_ID(_item(_hist,i)) =
                  the_ID(ida_id' POLL_LISTS_ID)
              )
            ),
        { all events in this sequence are get-next-to-poll }

        all i, j:

```

```

implies
(
  and (
    _inrange(_hist, i),
    _inrange(_hist, j)
  ),
  and (
    _lists_get_next_to_poll_primary_list(
      _lists_get_next_to_poll_extract(_item(_hist, i)))
    = _lists_get_next_to_poll_primary_list(
      _lists_get_next_to_poll_extract(
        _item(_hist, j))),
    _lists_get_next_to_poll_secondary_list(
      _lists_get_next_to_poll_extract(
        _item(_hist, i))) =
    _lists_get_next_to_poll_secondary_list(
      _lists_get_next_to_poll_extract(
        _item(_hist, j))),
    _lists_get_next_to_poll_subcycle_count(
      _lists_get_next_to_poll_extract(
        _item(_hist, i))) =
    _lists_get_next_to_poll_subcycle_count(
      _lists_get_next_to_poll_extract(
        _item(_hist, j)))
  )
)
{ all events in this sequence have same poll lists }
)
then
  and (
    { each primary node selected for polling at least once
      in p+c times }
    all i:
      implies
        (
          and (
            int'ge(i, 1),
            int'le(i, p)
          ),
          { 1 <= i <= p }
          some k:
            and (
              _inrange(_hist, k),
              _kind(_item(_hist, k)) =
                _event_kind'LISTS_GET_NEXT_TO_POLL_KIND,
              _passive_ID(_item(_hist, k)) =
                the_ID(ida_id'POLL_LISTS_ID),
              _lists_get_next_to_poll_node_list(
                _lists_get_next_to_poll_extract(
                  _item(_hist, k))) =
                _node_list_type'PRIMARY,
              _lists_get_next_to_poll_node_index(
                _lists_get_next_to_poll_extract(
                  _item(_hist, k))) = i
            )
          ),
          { at least c secondary nodes selected in p+c times }

```

```

int'ge(_length(secondary_nodes(_hist)), c),

{ if secondary node polled twice, every other secondary node
  polled at least once in between }
all i,j:
  implies
    (
      and (
        _inrange(_hist, i),
        _inrange(_hist, j),
        int'lt(i, j),
        _kind(_item(_hist, i)) =
          _event_kind'LISTS_GET_NEXT_TO_POLL_KIND,
        _passive_ID(_item(_hist, i)) =
          the_ID(ida_id'POLL_LISTS_ID),
        _lists_get_next_to_poll_node_list(
          _lists_get_next_to_poll_extract(
            _item(_hist, i))) =
          _node_list_type'SECONDARY,
        _kind(_item(_hist, j)) =
          _event_kind'LISTS_GET_NEXT_TO_POLL_KIND,
        _passive_ID(_item(_hist, j)) =
          the_ID(ida_id'POLL_LISTS_ID),
        _lists_get_next_to_poll_node_list(
          _lists_get_next_to_poll_extract(
            _item(_hist, j))) =
          _node_list_type'SECONDARY,
        _lists_get_next_to_poll_node_index(
          _lists_get_next_to_poll_extract(ITEM(_hist, j))) =
          _lists_get_next_to_poll_node_index(
            _lists_get_next_to_poll_extract(
              _item(_hist, i)))
        { poll of same node }
      ),

all k:
  implies
    (
      and (
        int'ge(k, 1), int'le(k, s)
      ),
      { 1 <= k <= s }

      some m:
        and (
          _inrange(_hist, m),
          int'lt(i, m),
          int'gt(j, m), { i < m < j }
          _kind(_item(_hist, m)) =
            _event_kind'LISTS_GET_NEXT_TO_POLL_KIND,
          _passive_ID(_item(_hist, m)) =
            the_ID(ida_id'POLL_LISTS_ID),
          _lists_get_next_to_poll_node_list(
            _lists_get_next_to_poll_extract(
              _item(_hist, m))) =
            _node_list_type'SECONDARY,
          _lists_get_next_to_poll_node_index(
            _lists_get_next_to_poll_extract(

```

```

        _item(_hist,m))) = k
        { poll k'th node in secondary list }
    }
    )
    )
else
    true
    { if poll lists have not remained constant long enough,
      cannot check anything }
end if

end poll_cycle_check

function choose_events_on_same_lists(_hist, _event): _event_sequence =
{ starting at end of hist, returns choose-node-to-poll-events as long
  as the poll lists remained unchanged, and a poll-list reset was not
  issued }
{ note: ignores starting/stopping of bus-controller mode; is this right ?? }
pre
    and (
        _kind(_event) = _event_kind'LISTS_GET_NEXT_TO_POLL_KIND,
        _passive_ID(_event) = the_ID(ida_id'POLL_LISTS_ID)
    )

measure ord'val(_length(_hist))
begin

    if _is_empty(_hist)
    then
        empty_sequence
    else
        if
            and (
                _kind(_right(_hist)) =
                    _event_kind'LISTS_GET_NEXT_TO_POLL_KIND,
                _passive_ID(_right(_hist)) = the_ID(ida_id'POLL_LISTS_ID0)
            )
        then
            if
                and (
                    _lists_get_next_to_poll_primary_list(
                        _lists_get_next_to_poll_extract(_right(_hist))) =
                        _lists_get_next_to_poll_primary_list(
                            _lists_get_next_to_poll_extract(_event)),
                    _lists_get_next_to_poll_secondary_list(
                        _lists_get_next_to_poll_extract(_right(_hist))) =
                        _lists_get_next_to_poll_secondary_list(
                            _lists_get_next_to_poll_extract(_event)),
                    _lists_get_next_to_poll_subcycle_count(
                        _lists_get_next_to_poll_extract(_right(_hist))) =
                        subcycle_count(_lists_get_next_to_poll_extract(
                            _event))
                )
                { lists are the same }
            )
        then
            _apr(choose_events_on_same_lists(_dropr(_hist), _event),
                _right(_hist))
        end if
    end if
end

```

```

else
    empty_sequence { stop - poll lists have changed }
end if
else { was not get-next-to-poll event }
    if
        and (
            _kind(_right(_hist)) =
                _event_kind' SUPER_CMD_GET_KIND,
            _passive_ID(_right(_hist)) =
                the_ID(ida_id'FROM_SUPER_KIND),
            message_kind(message(_right(_hist))) =
                _msg_kind_type'RESET
        )
    then
        empty_sequence { stop - supervisor reset has occurred }
    else
        choose_events_on_same_lists(_dropr(_hist), _event)
        { do not include this event but continue search }
    end if
end if
end if

end choose_events_on_same_lists

function check_poll_sequence(_hist, _event):bool =
    pre
        and (
            _kind(_event) = _event_kind' LISTS_GET_NEXT_TO_POLL_KIND,
            _passive_ID(_event) = the_ID(ida_id'POLL_LISTS_ID)
        )
    begin
        { if have chosen at least p+c nodes to poll, then check have
          chosen nodes to poll correctly }
        { note: check is on sequence consisting of event appended to hist }
        if
            int'ge(_length(_apr(choose_events_on_same_lists(_hist, _event),
                _event)),
                list_length(_lists_get_next_to_poll_primary_count(
                    _lists_get_next_to_poll_extract(_event)))
                + subcycle_count(_lists_get_next_to_poll_extract(_event)))
        then
            poll_cycle_check(
                _apr(choose_events_on_same_lists(_hist, _event),
                    _event),
                list_length(_lists_get_next_to_poll_primary_count(
                    _lists_get_next_to_poll_extract(_event))),
                list_length(data_secondary_count(
                    _lists_get_next_to_poll_extract(_event))),
                _lists_get_next_to_poll_subcycle_count(
                    extract_poll_list_data(_event)))
            else true
        end if
    end check_poll_sequence

function end_of_poll_cycle(_event): bool =
    { returns true if event ends a polling cycle }

```

```

begin
  or (
    { event was sending of select }
    and (
      _kind(_event) = _event_kind'ARB_CMD_PUT_KIND,
      _passive_ID(_event) = the_ID(ida_id'TO_CCI_ID),
      _arb_cmd_put_msg_kind(_arb_cmd_put_extract(_event)) =
        _msg_kind_type'SELECT
    ),

    { event was data channel not available }
    and (
      _kind(_event) = _event_kind'_STATE_OF_DATA_CHANNEL_GET_KIND,
      _passive_ID(_event) = the_ID(ida_id'DATA_CHANNEL_STATE_ID),
      time_for_select(_event) = false
    ),

    { event was get node on head of queue, and queues were empty }
    and (
      _kind(_event) = _event_kind'_QUEUES_GET_HEAD_KIND,
      _passive_ID(_event) = the_ID(ida_id'SELECT_QUEUES_ID),
      _return_value(_event) = _event_return'NO_MESSAGE
    )
  )
end end_of_poll_cycle

function choose_node_to_poll_seq_ok(_hist): bool =
begin
  { after end polling cycle, start over with selecting node to poll }
  all i:
    implies
      (
        and (
          _inrange(_hist, i),
          or (
            end_of_poll_cycle(_item(_hist, i)),
            starting_bus_controller_mode(
              first_n_events(_hist, minus(i, 1)),
              _item(_hist, i))
            { event is start of bus controller mode }
          )
        ),
        if _kind(next_arb_event(_hist, i)) <> _event_kind'NIL_EVENT
        then
          and (
            _kind(next_arb_event(_hist, i)) =
              _event_kind'_LISTS_GET_NEXT_TO_POLL_KIND,
            _passive_ID(next_arb_event(_hist, i)) =
              the_ID(ida_id'POLL_LISTS_ID)
          )
          else true
        end if
      )
  end
end choose_node_to_poll_seq_ok

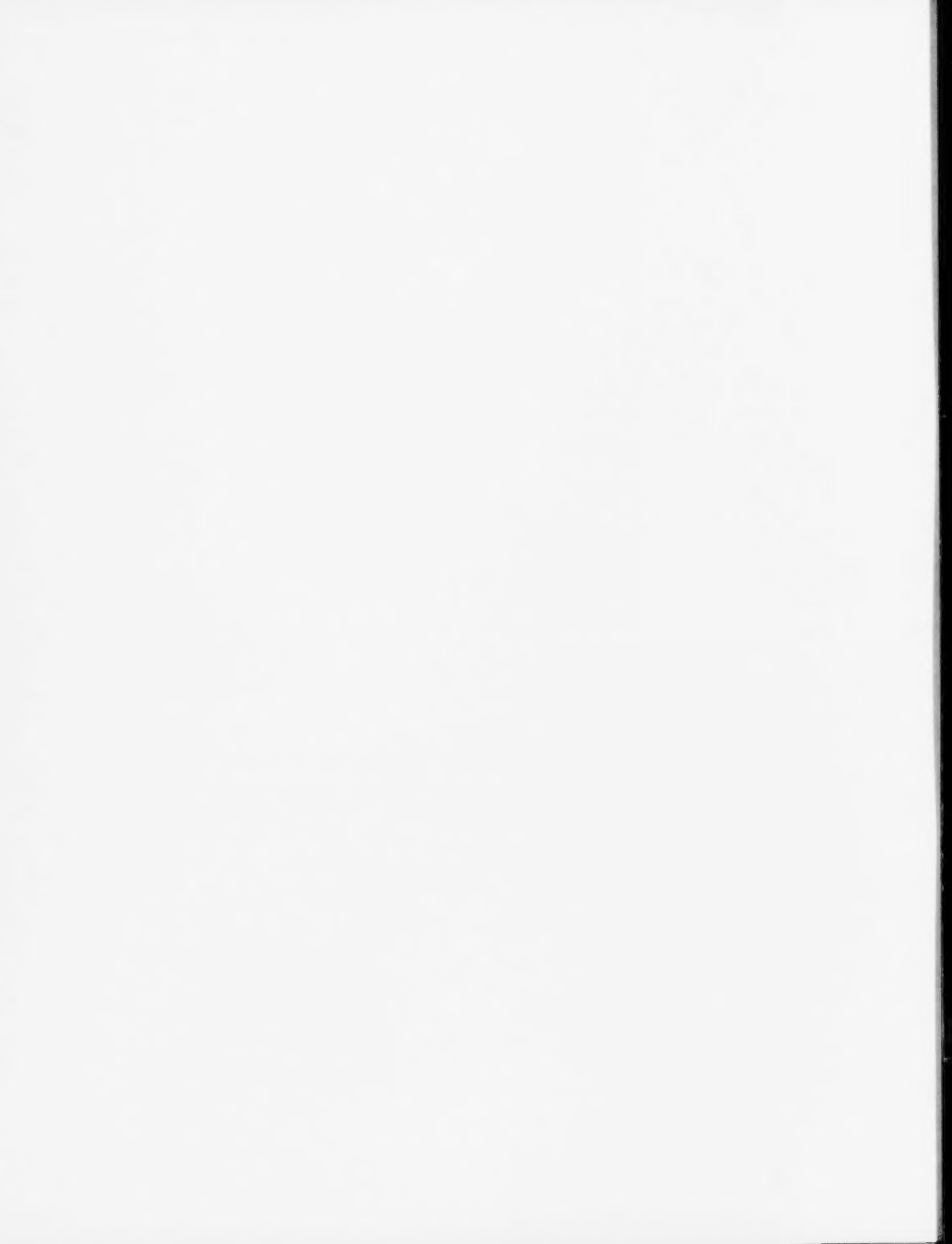
```

```

function choose_node_to_poll_append_event_ok(_hist, _event): bool =
  begin
    if
      and (
        _kind(_event) = _event_kind'LISTS_GET_NEXT_TO_POLL_KIND,
        _passive_ID(_event) = the_ID(ida_id'POLL_LISTS_ID)
      )
    then
      and (
        arb_event_valid(_hist, _event),
        { node in bus controller mode }
        _return_value(_event) = _event_return'OK,
        check_poll_sequence(_hist, _event),
        { polling nodes in right order }
        {
          if _length(arb_events(_hist)) = 0
          then true
          else
            end_of_poll_cycle(_right(arb_events(_hist)))
          end if
        }
        { this event is first in polling cycle }
        { could do here or up above }
        { here check that can only start if at end of cycle }
        { above check that have to start if end of cycle }
        { it is shown in both places to demonstarte possible
          approaches }
      )
    else true
    end if
  end choose_node_to_poll_append_event_ok

function ok_choose_node_to_poll(_hist): bool =
  measure ord'val(_length(_hist))
  begin
    if _is_empty(_hist)
    then true
    else
      and (
        choose_node_to_poll_seq_ok(_hist),
        choose_node_to_poll_append_event_ok(_dropr(_hist),
          _right(_hist)),
        ok_choose_node_to_poll(_dropr(_hist))
      )
    end if
  end ok_choose_node_to_poll

```



{ functions used to verify goodness with respect to events of kind clear
(of error pool) }

```
function error_clear_seq_ok(_hist): bool =  
  begin  
    true  
  end error_clear_seq_ok
```

```
function error_clear_append_event_ok(_hist, _event): bool =  
  begin  
    if  
      and (  
        _kind(_event) = _event_kind' ERRORS_CLEAR_KIND,  
        _passive_ID(_event) = the_ID(ida_id' ARB_ERRORS_ID)  
      )  
    then  
      _return_value(_event) = _event_return' OK  
    else true  
    end if  
  end error_clear_append_event_ok
```

```
function ok_error_clear(_hist): bool =  
  measure ord'val(_length(_hist))  
  begin  
    if _is_empty(_hist)  
    then true  
    else  
      and (  
        error_clear_seq_ok(_hist),  
        error_clear_append_event_ok(dropr(_hist), _right(_hist)),  
        ok_error_clear(dropr(_hist))  
      )  
    end ok_error_clear
```

```

{ functions used to verify goodness with respect to events of kind
  get-head (of select-queues) }

function data_channel_state_get_seq_ok(_hist): bool =
  begin
    { before select node from head of queue (in order to select it),
      must look at status of data channel }
    all i:
      implies
        (
          and (
            _inrange(_hist, i),
            _kind(_item(_hist, i)) =
              _event_kind' QUEUES_GET_HEAD_KIND,
            _passive_ID(_item(_hist, i)) =
              the_ID(ida_id'SELECT_QUEUES_ID)
          ),
          and (
            int'gt(_length(arb_events(_first_i_events(_hist, i-1))), 1)
              { must be in arbitration cycle }
            _kind(_right(arb_events(_first_i_events(_hist, i-1))) =
              _event_kind' STATE_OF_DATA_CHANNEL_GET_KIND,
            _passive_ID(_right(arb_events(_first_i_events(_hist, i-1)))
              = the_ID(ida_id'DATA_CHANNEL_ID),
            time_for_select(
              _right(arb_events(_first_i_events(_hist, i-1))),
              arb_events(_first_i_events(_hist, i-2)))
              = true
              { data channel has started to send previous message }
          )
        )
    end data_channel_state_get_seq_ok

function data_channel_state_get_append_event_ok(_hist, _event): bool =
  begin
    if
      and (
        _kind(_event) = _event_kind' STATE_OF_DATA_CHANNEL_GET_KIND,
        _passive_ID(_event) = the_ID(ida_id'DATA_CHANNEL_ID)
      )
    then
      and (
        arb_event_valid(_hist, _event), { in bus controller mode }
        _return_value(_event) = _event_return'OK
      )
    else true
    end if
  end data_channel_state_get_append_event_ok

function ok_data_channel_state_get(_hist): bool =
  measure ord'val(_length(_hist))
  begin
    if _is_empty(_hist)
    then true
    else
      and (

```

```
data_channel_state_get_seq_ok(_hist),  
data_channel_state_get_append_event_ok(  
    _dropr(_hist), _right(_hist)),  
ok_data_channel_state_get(_dropr(_hist))  
)  
end if  
end ok_data_channel_state_get
```



```
{ functions used to verify goodness with respect to events of kind
  write to error pool }
```

```
var node_id: _node_id__type
```

```
function request_denied(_hist, i, node_id): bool =
  { event i in hist is send of request denied to node node_id }
  begin
    and (
      _kind(_item(_hist, i)) = _event_kind'ARB_CMD_PUT_KIND,
      _passive_ID(_item(_hist, i)) = the_ID(ida_id'TO_CCI_ID),
      _arb_cmd_put_msg_kind(_arb_cmd_put_extract(_item(_hist,i))) =
        _msg_kind_type'REQUEST_DENIED,
      _arb_cmd_put_dest_node_id(_arb_cmd_put_extract(_item(_hist,i)))
        = node_id
    )
  end request_denied
```

```
function err_seq_ok(_hist): bool =
  begin
    and (
      { if poll request times out, write error }
      all i:
        implies
          (
            and (
              _inrange(_hist, i),
              _kind(_item(_hist, i)) =
                _event_kind'POLL_RESPONSE_GET_KIND,
              _passive_ID(_item(_hist, i)) =
                the_ID(ida_id'FROM_CCI_ID),
              _return_value(_item(_hist, i)) =
                _event_return'TIMED_OUT,
              int'gt(_length(_hist), i)
            ),
            and (
              _kind(_item(_hist, plus(i,1))) =
                _event_kind'ERRORS_WRITE_KIND,
              _passive_ID(_item(_hist, plus(i,1))) =
                the_ID(ida_id'ARB_ERRORS_ID),
              _errors_write_err_id(_errors_write_extract(
                _item(_hist, plus(i,1)))) =
                _err_id_type'POLL_TIMEOUT_ERR,
              _errors_write_node_id(_errors_write_extract(
                _item(_hist, plus(i,1)))) =
                node_response_get_source_node_id(
                  node_response_get_extract(_item(_hist, i)))
              { error says which node timed out }
            )
          )
      )
      { if attempt to put node on select queue fails because
        select queue full, write error, and send request denied }
      all i:
        implies
```

```

(
  and (
    _inrange(_hist, i),
    _kind(_item(_hist, i)) =
      _event_kind' QUEUES_PUT_NODE_KIND,
    _passive_ID(_item(_hist, i)) =
      the_ID(ida_id' SELECT_QUEUES_ID),
    _return_value(_item(_hist, i)) =
      _event_return' POOL_FULL,
    int'gt(_length(_hist), plus(i,1))
  ),
  and (
    _kind(_item(_hist, plus(i,1))) =
      _event_kind' ERRORS_WRITE_KIND,
    _passive_ID(_item(_hist, plus(i,1))) =
      the_ID(ida_id' ARB_ERRORS_ID),
    _errors_write_err_id(_errors_write_extract(_item(
      _hist, plus(i,1)))) =
      _err_id_type' QUEUE_OVERFLOW_ERR,
    _errors_write_queue(_errors_write_extract(_item(
      _hist, plus(i,1)))) =
      _queues_get_or_put_queue(
        _queues_get_or_put_extract(_item(_hist, i))),
    { error says which queue was full }
    request_denied(_hist, plus(i,2),
      _queues_get_or_put_node_id(
        _queues_get_or_put_extract(_item(_hist, i))))
    { after write error, send request denied to node
      which sent poll response }
  )
)

,

{ if node fails to send after selected, write error }
all i:
  implies
  (
    and (
      _inrange(_hist, i),
      _kind(_item(_hist, i)) =
        _event_kind' STATE_OF_DATA_CHANNEL_GET_KIND,
      _passive_ID(_item(_hist, i)) =
        the_ID(ida_id' DATA_CHANNEL_STATE_ID),
      node_failed_to_send(_first_i_events(_hist, minus(i,1)),
        _item(_hist, i))
    ),
    and (
      _kind(_item(_hist, plus(i,1))) =
        _event_kind' ERRORS_WRITE_KIND,
      _passive_ID(_item(_hist, plus(i,1))) =
        the_ID(ida_id' ARB_ERRORS_ID),
      _errors_write_err_id(_errors_write_extract(_item(
        _hist, plus(i,1)))) =
        _err_id_type' NODE_FAILED_TO_SEND,
      _errors_write_node_id(_errors_write_extract(_item(
        _hist, plus(i,1)))) =
        _arb_cmd_put_dest_node_id(
          _error_report_write_extract(

```

```

        last_cci_msg_sent(
            _first_i_events(_hist, minus(i,1)),
            SELECT)))
        { node which failed to send }
    ),
),
{ if node sends for too long after selected, write error }
all i:
    implies
    (
        and (
            _inrange(_hist, i),
            _kind(_item(_hist, i)) =
                _event_kind' STATE_OF_DATA_CHANNEL_GET_KIND,
            _passive_ID(_item(_hist, i)) =
                the_ID(ida_id'DATA_CHANNEL_STATE_ID),
            node_sent_too_long(_first_i_events(_hist, minus(i,1)),
                _item(_hist, i))
        ),
        and (
            _kind(_item(_hist, plus(i,1))) =
                _event_kind' ERRORS_WRITE_KIND,
            _passive_ID(_item(_hist, plus(i,1))) =
                the_ID(ida_id'ARB_ERRORS_ID),
            _errors_write_err_id(_errors_write_extract(_item(
                _hist, plus(i,1)))) =
                _err_id_type'NODE_MSG_TOO_LONG,
            _errors_write_node_id(_errors_write_extract(_item(
                _hist, plus(i,1)))) =
                _arb_cmd_put_dest_node_id(_arb_cmd_put_extract(
                    last_cci_msg_sent(
                        _first_i_events(_hist, minus(i,1)),
                        SELECT)))
            { node which sent too long }
        )
    )
)
end err_seq_ok

```

```

function err_append_event_ok(_hist, _event): bool =
begin
    { note: do not insist that node be bus controller to write error }
    { may want to change this }
    if
        and (
            _kind(_event) = _event_kind' ERRORS_WRITE_KIND,
            _passive_ID(_event) = the_ID(ida_id'ARB_ERRORS_ID)
        )
    then
        and (
            or (
                _errors_write_err_id(_errors_write_extract(_event)) =
                    _err_id_type'POLL_TIMEOUT_ERR,
                _errors_write_err_id(_errors_write_extract(_event)) =
                    _err_id_type'QUEUE_OVERFLOW_ERR,
                _errors_write_err_id(_errors_write_extract(_event)) =

```

```
        _err_id_type'NODE_FAILED_TO_SEND,
        _errors_write_err_id(_errors_write_extract(_event)) =
        _err_id_type'NODE_MSG_TOO_LONG
    )
    _return_value(_event) = _event_return'OK,
)
else true
end if

end err_append_event_ok

function ok_err(_hist): bool =
measure ord'val(_length(_hist))
begin
    if _is_empty(_hist)
    then true
    else
        and (
            err_seq_ok(_hist),
            err_append_event_ok(_dropr(_hist), _right(_hist)),
            ok_err(_dropr(_hist))
        )
    end if
end ok_err
```

```
{ functions to verify goodness with respect to events of kind
  read from error pool }
```

```
function errors_get_seq_ok(_hist): bool =
  begin
    true
    { send_errors_seq_ok checks that super error request causes error get }
  end errors_get_seq_ok
```

```
function errors_get_append_event_ok(_hist, _event): bool =
  begin
    if
      and (
        _kind(_event) = _event_kind' ERRORS_READ_KIND,
        _passive_ID(_event) = the_ID(ida_id' ARB_ERRORS_ID)
      )
    then
      _return_value(_event) = _event_return' OK
      { Note: it is assumed that if the error pool is empty, then a
        report saying so will still be returned }
    else true
    end if
  end errors_get_append_event_ok
```

```
function ok_errors_get(_hist): bool =
  measure ord'val(length(_hist))
  begin
    if _is_empty(_hist)
    then true
    else
      and (
        errors_get_seq_ok(_hist),
        errors_get_append_event_ok(dropr(_hist), _right(_hist)),
        ok_errors_get(dropr(_hist))
      )
    end ok_errors_get
```

```
{ functions to verify goodness with respect to events of kind
  get-node-state (from node state pool ) }
```

```
const STATE_PERIOD: int
  { frequency with which must read node state }
```

```
function node_state_seq_ok(_hist): bool =
  begin
    and (
      { first event is look at node status }
      if
        int'ge(_length(_hist), 0)
      then
        and(
          _kind(_left(_hist)) = _event_kind'_STATE_OF_NODE_GET_KIND,
          _passive_ID(_left(_hist)) = the_ID(ida_id'NODE_STATE_ID),
        )
      else true
      end if,

      { periodically read node state to see if bus controller }
      { i.e. if two events are sufficiently far apart in time, there is
        a read of node state event between them }
      all i, j:
        implies
          (
            and(
              _inrange(_hist, i),
              _inrange(_hist, j),
              int'ge (
                minus(_time(_item(_hist, j)),
                  _time(_item(_hist, i))),
                STATE_PERIOD
              )
            { event j occurred at least STATE_PERIOD ticks after
              event i }
            ),
          some k:
            and (
              and (
                int'ge(k,i), int'le(k,j)
              ),
              { i <= k <= j }
              _kind(_item(_hist, k)) =
                _event_kind'_STATE_OF_NODE_GET_KIND,
              _passive_ID(_item(_hist, k)) =
                the_ID(ida_id'NODE_STATE_ID)
            )
          )
        )
    end node_state_seq_ok
```

```
function node_state_append_event_ok(_hist, _event): bool =
  begin
    if
      and (
```

```

        _kind(_event) = _event_kind' STATE_OF_NODE_GET_KIND,
        _passive_ID(_event) = the_ID(ida_id'NODE_STATE_ID)
    )
    then
        and (
            or (
                _state_of_node_get_state(_state_of_node_get_extract(
                    _event)) =
                    _node_state_type'IS_BUS_CONTROLLER,
                _state_of_node_get_state(_state_of_node_get_extract(
                    _event)) =
                    _node_state_type'IS_NOT_BUS_CONTROLLER,
            ),
            _return_value(_event) = _event_return'OK
        )
    else true
    end if

end node_state_append_event_ok

function ok_node_state(_hist): bool =
    measure ord'val(_length(_hist))
begin
    if _is_empty(_hist)
    then true
    else
        and (
            node_state_seq_ok(_hist),
            node_state_append_event_ok(_dropr(_hist), _right(_hist)),
            ok_node_state(_dropr(_hist))
        )
    end if
end ok_node_state

```

```

{ functions to verify goodness with respect to events of kind
  get-poll-response (from channel to cci) }

const CYCLE_TIME: int
  ( when get poll response, must send another arbitration command this fast )

function in_same_cycle(_hist, i, j): bool =
begin
  { check that do not stop arbitrating between events i and j }
  all k:
    implies
      (
        and (
          int'gt(k,i), int'lt(k,j)
        ),
        if
          and (
            _kind(_item(_hist, k)) =
              _event_kind' STATE_OF_NODE_GET_KIND,
            _passive_ID(_item(_hist, k)) =
              the_ID(ida_id'NODE_STATE_ID)
          )
        then
          event_is_bus_controller(_item(_hist, k))
        else
          true
        end if
      )
end in_same_cycle

function get_poll_response_seq_ok(_hist): bool =
begin
  and (
    { poll cycle is fast enough i.e. if event i if getting of poll response,
      then within CYCLE_TIME must send another arbitration command }
    all i,j:
      implies
        (
          and (
            _inrange(_hist, i),
            _inrange(_hist, j),
            int'ge(
              minus(_time(_item(_hist, j)), _time(_hist, i)),
              CYCLE_TIME
            )
          )
          _kind(_item(_hist, i)) =
            _event_kind' POLL_RESPONSE_GET_KIND,
          _passive_ID(_item(_hist, i)) = the_ID(ida_id'FROM_CCI_ID),
          in_same_cycle(_hist, i, j)
            { events i and j in same arbitration sequence }
        ),
    some k:
      { must be event between which is sending of arb. command }
      and (
        and (
          int'gt(k, i), int'le(k, j)
          { i < k <= j }

```

```

    )
    _passive_ID(_item(_hist, k)) =
        the_ID(ida_id'TO_CCI_ID),
    _kind(_item(_hist, k)) = _ARB_CMD_PUT_KIND
    )
),

{ after send poll request to node, get that node's response (if still
  in bus controller mode )
all i:
    implies
    (
        and (
            _inrange(_hist, i),
            _kind(_item(_hist, i)) =
                _event_kind' POLL_RESPONSE_GET_KIND,
            _passive_ID(_item(_hist, i)) = the_ID(ida_id'FROM_CCI_ID)
        ),
        and (
            arb_events(_first_i_events(_hist, minus(i,1))) <>
                empty_sequence,
            { must be in midst of arbitration cycle }
            _kind(_right(arb_events(_hist, minus(i,1)))) =
                _event_kind' ARB_CMD_PUT_KIND,
            _passive_ID(_right(arb_events(_hist, minus(i,1))) =
                the_ID(ida_id'TO_CCI_ID),
            _arb_cmd_put_msg_kind(_arb_cmd_put_extract(_right(
                arb_events(_hist, minus(i,1)))) =
                _msg_kind_type' POLL_REQUEST,
            _arb_cmd_put_dest_node_id(_arb_cmd_put_extract(
                _right(arb_events(_hist, minus(i,1)))) =
                _node_response_get_source_node_id(
                    _node_response_get_extract(_item(_hist, i)))
            { response is from node which was queried }
        )
    )
)

end get_poll_response_seq_ok

function get_poll_response_append_event_ok(_hist, _event): bool =
begin
    if
        and (
            _kind(_event) = _event_kind' POLL_RESPONSE_GET_KIND,
            _passive_ID(_event) = the_ID(ida_id'FROM_CCI_ID)
        )
    then
        and (
            arb_event_valid(_hist, _event), { in bus controller mode }
        or
            and (
                _return_value(_event) = _event_return'OK,
            or
                _node_response_get_response_kind(
                    _node_response_get_extract(_event)) =
                    _response_type'HAS_MESSAGE,
            )
        )
    end if
end

```

```
        _node_response_get_response_kind(
            _node_response_get_extract(_event)) =
            _response_type'HAS_NO_MESSAGE
    ),
    _return_value(_event) = _event_return'TIMED_OUT
)
)
else true
end if

end get_poll_response_append_event_ok

function ok_get_poll_response(_hist): bool =
    measure ord'val(_length(_hist))
begin
    if _is_empty(_hist)
    then true
    else
        and (
            get_poll_response_seq_ok(_hist),
            get_poll_response_append_event_ok(_dropr(_hist), _right(_hist)),
            ok_get_poll_response(_dropr(_hist))
        )
    end if
end ok_get_poll_response
```

```
{ functions to verify goodness with respect to events of kind add-node
  (to select queues) }
```

```
function queue_add_node_seq_ok(_hist): bool =
  begin
    { after get poll response, queue message }
    all i:
      implies
        (
          and (
            _inrange(_hist, i),
            _kind(_item(_hist, i)) =
              _event_kind' POLL_RESPONSE_GET_KIND,
            _passive_ID(_item(_hist, i)) = the_ID(ida_id'FROM_CCI_ID),
            _return_value(_item(_hist, i)) = _event_return'OK ,
            { did not time out }
            _node_response_get_response_kind(
              _node_response_get_extract(_item(_hist, i))) =
              _response_type'HAS_MESSAGE
          ),
          if
            _kind(next_arb_event(_hist, i)) <> _event_kind'NIL_EVENT
          then
            and (
              _kind(next_arb_event(_hist, i)) =
                _event_kind' QUEUES_PUT_NODE_KIND,
              _passive_ID(next_arb_event(_hist, i)) =
                the_ID(ida_id'SELECT_QUEUES_ID),
              _queues_get_or_put_node_id(
                _queues_get_or_put_extract(
                  next_arb_event(_hist, i))) =
                _node_response_get_source_node_id(
                  _node_response_get_extract(
                    _item(_hist, i))),
              { queue node which repoded }
              _queues_get_or_put_queue(_queues_get_or_put_extract(
                next_arb_event(_hist, i))) =
                _node_response_get_priority(
                  _node_response_get_extract(_item(_hist, i)))
            )
          else
            true
          end if
        )
      end queue_add_node_seq_ok
```

```
function queue_add_node_append_event_ok(_hist, _event): bool =
  begin
    if
      and (
        _kind(_event) = _event_kind' QUEUES_PUT_NODE_KIND,
        _passive_ID(_event) = the_ID(ida_id'SELECT_QUEUES_ID)
      )
    then
      and (
        arb_event_valid(_hist, _event), { in bus controller mode }
        or (
```

```
        _return_value(_event) = _event_return'OK,  
        _return_value(_event) = _event_return'POOL_FULL  
    )  
    else true  
    end if  
  
end queue_add_node_append_event_ok  
  
function ok_queue_add_node(_hist): bool =  
    measure ord'val(_hist)  
    begin  
        if _is_empty(_length(_hist))  
        then true  
        else  
            and (  
                queue_add_node_seq_ok(_hist),  
                queue_add_node_append_event_ok(_dropr(_hist), _right(_hist)),  
                ok_queue_add_node(_dropr(_hist))  
            )  
        end if  
    end  
end ok_queue_add_node
```

```
{ functions to verify goodness with respect to events of kind
  get-head (of select queues) }
```

```
function queue_select_head_seq_ok(_hist): bool =
  begin
    { if get head of queues and queues not empty,
      next event is sending of select to node at
      head of queue }
    all i:
      implies
        (
          and (
            _inrange(_hist, i),
            _kind(_item(_hist, i)) =
              _event_kind' QUEUES_GET_HEAD_KIND,
            _passive_ID(_item(_hist, i)) =
              the_ID(ida_id' SELECT_QUEUES_ID),
            _return_value(_item(_hist, i)) <> _event_return' POOL_EMPTY
          ),
          if
            _kind(next_arb_event(_hist, i)) <> _event_kind' NIL_EVENT
          then
            and (
              _kind(next_arb_event(_hist, i)) =
                _event_kind' ARB_CMD_PUT_KIND,
              _passive_ID(next_arb_event(_hist, i)) =
                the_ID(ida_id' TO_CCI_ID),
              _arb_cmd_put_msg_kind(
                _arb_cmd_put_extract(next_arb_event(_hist, i))) =
                _msg_kind_type' SELECT,
              _arb_cmd_put_dest_node_id(_arb_cmd_put_extract(
                next_arb_event(_hist, i))) =
                _queues_get_or_put_type(
                  _queues_get_or_put_extract(_item(_hist, i)))
              { send select to node at head of queue }
            )
          else
            true
          end if
        )
      end queue_select_head_seq_ok
```

```
function queue_select_head_append_event_ok(_hist, _event): bool =
  begin
    if
      and (
        _kind(_event) = _event_kind' QUEUES_GET_HEAD_KIND,
        _passive_ID(_event) = the_ID(ida_id' SELECT_QUEUES_ID)
      )
    then
      and (
        arb_event_valid(_hist, _event), { in bus controller mode }
        or (
          _return_value(_event) = _event_return' OK,
          _return_value(_event) = _event_return' POOL_EMPTY
        )
      )
    )
```

```
        else true
        end if

    end queue_select_head_append_event_ok

function ok_queue_select_head(_hist): bool =
    measure ord'val(_length(_hist))
begin
    if _is_empty(_hist)
    then true
    else
        and (
            queue_select_head_seq_ok(_hist),
            queue_select_head_append_event_ok(_dropr(_hist), _right(_hist)),
            ok_queue_select_head(_dropr(_hist))
        )
    end if
end ok_queue_select_head
```

```
{ functions to verify goodness with respect to events of kind
  remove-node (from select queues) }
```

```
function queue_remove_node_seq_ok(_hist): bool =
  begin
    { if select node, remove it from queue }
    all i:
      implies
        (
          and (
            _inrange(_hist, i),
            _kind(_item(_hist, i)) = _event_kind'ARB_CMD_PUT_KIND,
            _passive_ID(_item(_hist, i)) = the_ID(ida_id'TO_CCI_ID),
            _arb_cmd_put_msg_kind(_arb_cmd_put_extract(
              _item(_hist, i))) =
              _msg_kind_type'SELECT,
            _return_value(_item(_hist, i)) = _event_return'OK
          ),
          if
            _kind(next_arb_event(_hist, i)) <> _event_kind'NIL_EVENT
          then
            and (
              _kind(next_arb_event(_hist, i)) =
                _event_kind'QUEUES_REMOVE_NODE_KIND,
              _passive_ID(next_arb_event(_hist, i)) =
                the_ID(ida_id'SELECT_QUEUES_ID),
              _queues_get_or_put_node_id(
                _queues_get_or_put_extract(
                  next_arb_event(_hist, i))) =
                _arb_cmd_put_dest_node_id(
                  _arb_cmd_put_extract(_item(_hist, i))),
              { remove node which was selected }
              _queues_get_or_put_queue(_queues_get_or_put_extract(
                next_arb_event(_hist, i))) =
                _queues_get_or_put_queue(
                  _queues_get_or_put_extract(_last_event(
                    _first_i_events(_hist, i-1),
                    _event_kind'QUEUES_GET_HEAD_KIND,
                    theID(ida_id'SELECT_QUEUES_ID))))
              { remove from queue from which we got it }
            )
          else
            true
          end if
        )
      end queue_remove_node_seq_ok
```

```
function queue_remove_node_append_event_ok(_hist, _event): bool =
  begin
    if
      and (
        _kind(_event) = _event_kind'QUEUES_REMOVE_NODE_KIND,
        _passive_ID(_event) = the_ID(ida_id'SELECT_QUEUES_ID)
      )
    then
      and (
```

```
        arb_event_valid(_hist, _event), { in bus controller mode }
        _return_value(_event) = _event_return'OK,
    )
    else true
    end if

end queue_remove_node_append_event_ok

function ok_queue_remove_node(_hist): bool =
    measure ord'val(_length(_hist))
begin
    if _is_empty(_hist)
    then true
    else
        and (
            queue_remove_node_seq_ok(_hist),
            queue_remove_node_append_event_ok(_dropr(_hist), _right(_hist)),
            ok_queue_remove_node(_dropr(_hist))
        )
    end if
end ok_queue_remove_node
```

```
{ functions to verify goodness with respect to events of kind
  reset of poll-lists }
```

```
function poll_reset_seq_ok(_hist): bool =
  begin
    { supervisor reset always followed by poll-list reset and error-clear }
    all i:
      implies
        (
          and (
            _inrange(_hist, i),
            _kind(_item(_hist, i)) = _event_kind'M_SUPER_CMD_GET_KIND,
            _passive_ID(_item(_hist, i)) =
              the_ID(ida_id'FROM_SUPER_ID),
            _super_cmd_get_command_kind(
              _super_cmd_get_extract(_item(_hist, i))) =
              _msg_command_type'RESET,
            int'gt(_length(_hist), plus(i, 1))
          ),
          or (
            and (
              _kind(_item(_hist, plus(i,1))) =
                _event_kind'LISTS_RESET_KIND,
              _passive_ID(_item(_hist, plus(i,1))) =
                the_ID(ida_id'POLL_LISTS_ID),
              _kind(_item(_hist, plus(i,2))) =
                _event_kind'ERRORS_CLEAR_KIND,
              _passive_ID(_item(_hist, plus(i, 2))) =
                the_ID(ida_id'ARB_ERRORS_ID)
            ),
            and (
              _kind(_item(_hist, plus(i,1))) =
                _event_kind'ERRORS_CLEAR_KIND,
              _passive_ID(_item(_hist, plus(i, 1))) =
                the_ID(ida_id'ARB_ERRORS_ID),
              _kind(_item(_hist, plus(i,2))) =
                _event_kind'LISTS_RESET_KIND,
              _passive_ID(_item(_hist, plus(i,2))) =
                the_ID(ida_id'POLL_LISTS_ID)
            )
          )
        )
      )
    end poll_reset_seq_ok
```

```
function poll_reset_append_event_ok(_hist, _event): bool =
  begin
    if
      and (
        _kind(_event) = _event_kind'LISTS_RESET_KIND,
        _passive_ID(_event) = the_ID(ida_id'POLL_LISTS_ID)
      )
    then
      _return_value(_event) = _event_return'OK
    else true
    end if
```

```
end poll_reset_append_event_ok

function ok_poll_reset(_hist): bool =
  measure ord'val(_length(_hist))
begin
  if _is_empty(_hist)
  then true
  else
    and (
      poll_reset_seq_ok(_hist),
      poll_reset_append_event_ok(_dropr(_hist), _right(_hist)),
      ok_poll_reset(_dropr(_hist))
    )
  end if
end ok_poll_reset
```

```
{ functions to verify goodness with respect to events of kind
  write of error report (to supervisor) }
```

```
function send_errors_seq_ok(_hist): bool =
```

```
  begin
```

```
    { immediately after get request from supervisor for error report,
      get errors from pool and send them to supervisor, and then
      clear error pool }
```

```
    all i:
```

```
      implies
```

```
      (
```

```
        and (
```

```
          _inrange(_hist, i),
          _kind(_item(_hist, i)) = _event_kind'_SUPER_CMD_GET_KIND,
          _passive_ID(_item(_hist, i)) =
            the_ID(ida_id'FROM_SUPER_ID),
          _super_cmd_get_command_kind(_super_cmd_get_extract(
            _item(_hist, i))) =
            _msg_command_type'ERROR_REQUEST,
          _return_value(_item(_hist, i)) = _event_return'OK,
          int'gt(_length(_hist), plus(i, 2))
        ),
```

```
        and (
```

```
          _kind(_item(_hist, plus(i,1))) =
            _event_kind'_ERRORS_READ_KIND,
          _passive_ID(_item(_hist, plus(i,1))) =
            the_ID(ida_id'ARB_ERRORS_ID),
          _kind(_item(_hist, plus(i,2))) =
            _event_kind'_ERROR_REPORT_WRITE_KIND,
          _passive_ID(_item(_hist, plus(i, 2))) =
            the_ID(ida_id'TO_SUPER_ID),
          _errors_read_err_report(_errors_read_extract(
            _item(_hist, plus(i,1)))) =
            _error_report_write_report(
              _error_report_write_extract(
                _item(_hist, plus(i,2)))),
          { send to supervisor same error report which
            was obtained from error pool }
          _kind(_item(_hist, plus(i,3))) =
            _event_kind'_ERRORS_CLEAR_KIND,
          _passive_ID(_item(_hist, plus(i, 3))) =
            the_ID(ida_id'ARB_ERRORS_ID)
        )
      )
    )
```

```
  end send_errors_seq_ok
```

```
function send_errors_append_event_ok(_hist, _event): bool =
```

```
  begin
```

```
    if
```

```
      and (
```

```
        _kind(_event) = _event_kind'_ERROR_REPORT_WRITE_KIND,
        _passive_ID(_event) = the_ID(ida_id'TO_SUPER_ID)
      )
```

```
    then
```

```
      _return_value(_event) = _event_return'OK
```

```
    else true
```

```
    end if
```

```
end send_errors_append_event_ok

function ok_send_errors(_hist): bool =
  measure ord'val(_length(_hist))
begin
  if _is_empty(_hist)
  then true
  else
    and (
      send_errors_seq_ok(_hist),
      send_errors_append_event_ok(_dropr(_hist), _right(_hist)),
      ok_send_errors(_dropr(_hist))
    )
  end if
end ok_send_errors
```

```
{ functions to verify goodness with respect to events of kind
  get-supervisor-command }
```

```
const SUPER_PERIOD: int
  { frequency with which must read supervisor commands }
```

```
function super_seq_ok(_hist): bool =
  begin
    { periodically read supervisor commands }
    { i.e. if two events are sufficiently far apart in time, there is a
      read form the supervisor command channel between them }
    { Note: happens whether in bus controller mode or not }
    all i, j:
      implies
        (
          and (
            _inrange(_hist, i),
            _inrange(_hist, j),
            int'ge (
              minus(_time(_item(_hist, j)), _time(_item(_hist, i))),
              SUPER_PERIOD
            )
            { event j occurred at least SUPER_PERIOD ticks after
              event i }
          ),
          some k:
            and (
              and (
                int'ge(k, i), int'le(k, j)
              ),
              { i <= k <= j }
              _kind(_item(_hist, k)) =
                _event_kind' SUPER_CMD_GET_KIND,
              _passive_ID(_item(_hist, k)) =
                the_ID(ida_id'FROM_SUPER_ID)
            )
          )
        )
    end super_seq_ok
```

```
function super_append_event_ok(_hist, _event): bool =
  begin
    if
      and (
        _kind(_event) = _event_kind' SUPER_CMD_GET_KIND,
        _passive_ID(_event) = the_ID(ida_id'FROM_SUPER_ID)
      )
    then
      { two kinds of command from supervisor }
      if _return_value(_event) = _event_return'NO_MESSAGE
      then true
      else
        and (
          _return_value(_event) = _event_return'OK,
          or (
            _super_cmd_get_command_kind(
              _super_cmd_get_extract(_event)) =
```

```
        _msg_command__type'RESET,
        _super_cmd__get__command_kind(
            _super_cmd__get__extract(_event)) =
            _msg_command__type'ERROR_REQUEST
    )
end if
else true
end if

end super_append_event_ok

function ok_super(_hist): bool =
    measure ord'val(length(_hist))
begin
    if _is_empty(_hist)
    then true
    else
        and (
            super_seq_ok(_hist),
            super_append_event_ok(dropr(_hist), _right(_hist)),
            ok_super(dropr(_hist))
        )
    end if
end ok_super
```

```
function good(_external_spec): bool =
  begin
    correct(_history(_external_spec, myself))
  end good

function correct(_hist): bool =
  begin
    if _is_empty(_hist)
    then true
    else
      and (
        ok_node_state(_hist),
        ok_super(_hist),
        ok_poll_reset(_hist),
        ok_err(_hist),
        ok_send_errors(_hist),
        ok_error_clear(_hist),
        ok_choose_node_to_poll(_hist),
        ok_send_arb_msg(_hist),
        ok_get_poll_response(_hist),
        ok_queue_add_node(_hist),
        ok_queue_select_head(_hist),
        ok_queue_remove_node(_hist),
        ok_data_channel_state_get(_hist),
        ok_errors_get(_hist)
      )
    end if
  end correct
```

```
{ access procedures for channel from supervisor }
```

```
prog procedure _super_cmd_get(  
  mvar _external_spec,  
  mvar from_super_port,  
  pvar _a_command_kind,  
  pvar _err) =  
  
  initial (  
    _external_spec'0 = _external_spec,  
    from_super_port'0 = from_super_port  
  )  
  
  pre and (  
    good(_external_spec),  
    _well_sequenced(_external_spec)  
  )  
  
  post and (  
    _well_sequenced(_external_spec),  
    _socket_passive_ID(from_super_port) =  
      _socket_passive_ID(from_super_port'0),  
    all _ID_type'dummy:  
      _valid_active_ID(from_super_port, _ID_type'dummy)  
        = _valid_active_ID(from_super_port'0,  
          _ID_type'dummy),  
    some _event:  
      and (  
        _passive_ID(_event) =  
          _socket_passive_ID(from_super_port),  
        _valid_active_ID(from_super_port,  
          _active_ID(_event)),  
        int'ge(_get_time(_event),  
          _current_time(_external_spec)),  
        _kind(_event) =  
          _event_kind'_SUPER_CMD_GET_KIND,  
        _history(_external_spec, myself)  
          = _apr(_history(_external_spec'0, myself),  
            _event),  
        _return_value(_event) = _err,  
        _super_cmd_get_extract(_event)  
          = _super_cmd_get_type'val(  
            _a_command_kind)  
      )  
  )  
)
```

```
{ access procedures for channel to supervisor }
```

```
prog procedure _error_report__write(  
  mvar _external_spec,  
  mvar to_super__port,  
  lvar an_err_report,  
  pvar _err) =  
  
  initial (  
    _external_spec'0 = _external_spec,  
    to_super__port'0 = to_super__port  
  )  
  
  pre and (  
    good(_external_spec),  
    _well_sequenced(_external_spec)  
  )  
  post and (  
    _well_sequenced(_external_spec),  
    _socket_passive_ID(to_super__port) =  
      _socket_passive_ID(to_super__port'0),  
    all _ID_type'dummy:  
      _valid_active_ID(to_super__port, _ID_type'dummy)  
        = _valid_active_ID(to_super__port'0,  
          _ID_type'dummy),  
    some _event:  
      and (  
        _passive_ID(_event) =  
          _socket_passive_ID(to_super__port),  
        _valid_active_ID(to_super__port,  
          _active_ID(_event)),  
        int'ge(_get_time(_event),  
          _current_time(_external_spec)),  
        _kind(_event) =  
          _event_kind'_SUPER_CMD_GET_KIND,  
        _history(_external_spec, myself)  
          = _apr(_history(_external_spec'0, myself),  
            _event),  
        _return_value(_event) = _err,  
        _error_report__write__extract(event) ==  
          _error_report__write__type 'val(  
            an_err_report)  
      )  
  )  
)
```



```
{ access procedures for channel to control channel interface }
```

```
prog procedure _arb_cmd_put(  
  mvar _external_spec,  
  mvar to_cci_port,  
  lvar a_dest_node_id,  
  lvar a_msg_kind,  
  lvar a_priority,  
  pvar _err) =  
  
  initial (  
    _external_spec'0 = _external_spec,  
    to_cci_port'0 = to_cci_port  
  )  
  
  pre and (  
    good(_external_spec),  
    _well_sequenced(_external_spec)  
  )  
  
  post and (  
    _well_sequenced(_external_spec),  
    _socket_passive_ID(to_cci_port) =  
      _socket_passive_ID(to_cci_port'0),  
    all _ID_type'dummy:  
      _valid_active_ID(to_cci_port, _ID_type'dummy)  
        = _valid_active_ID(to_cci_port'0,  
          _ID_type'dummy),  
    some _event:  
      and (  
        _passive_ID(_event) =  
          _socket_passive_ID(to_cci_port),  
        _valid_active_ID(to_cci_port,  
          _active_ID(_event)),  
        int'ge(_get_time(_event),  
          _current_time(_external_spec)),  
        _kind(_event) =  
          _event_kind'_SUPER_CMD_GET_KIND,  
        _history(_external_spec, myself)  
          = _apr(_history(_external_spec'0, myself),  
            _event),  
        _return_value(_event) = _err,  
        _arb_cmd_put_extract(event)  
          = _arb_cmd_put_type'val(  
            a_dest_node_id, a_msg_kind),  
        if a_msg_kind = _msg_kind_type'SELECT  
        then  
          and (  
            _select_message_id(event) = a_msg_kind,  
            _select_priority(event) = a_priority  
          )  
        endif  
      )  
  )  
)
```

```
{ access procedures for node state pool }
```

```
prog procedure _state_of_node__get(  
    mvar _external_spec,  
    mvar node_state_pool,  
    pvar a_node_state,  
    pvar _err) =  
  
    initial (  
        _external_spec'0 = _external_spec,  
        node_state__port'0 = node_state__port  
    )  
  
    pre and (  
        good(_external_spec),  
        _well_sequenced(_external_spec)  
    )  
    post and (  
        _well_sequenced(_external_spec),  
        _socket_passive_ID(node_state__port) =  
            _socket_passive_ID(node_state__port'0),  
        all _ID_type'dummy:  
            _valid_active_ID(node_state__port, _ID_type'dummy)  
                = _valid_active_ID(node_state__port'0,  
                    _ID_type'dummy),  
        some _event:  
            and (  
                _passive_ID(_event) =  
                    _socket_passive_ID(node_state__port),  
                _valid_active_ID(node_state__port,  
                    _active_ID(_event)),  
                int'ge(_get_time(_event),  
                    _current_time(_external_spec)),  
                _kind(_event) =  
                    _event_kind' SUPER_CMD_GET_KIND,  
                _history(_external_spec, myself)  
                    = _apr(_history(_external_spec'0, myself),  
                        _event),  
                _return_value(_event) = _err,  
                _state_of_node__get_extract(event)  
                    = _state_of_node__get_type'val( a_node_state)  
            )  
    )  
)
```

{ access procedures for data channel state pool }

```
prog procedure _state_of_data_channel__get(  
  mvar _external_spec,  
  mvar data_channel_state__port,  
  pvar a_data_channel_state,  
  pvar a_data_channel_msg_id,  
  pvar _err) =  
  
  initial (  
    _external_spec'0 = _external_spec,  
    data_channel_state__port'0 = data_channel_state__port  
  )  
  
  pre and (  
    good(_external_spec),  
    _well_sequenced(_external_spec)  
  )  
  
  post and (  
    _well_sequenced(_external_spec),  
    _socket_passive_ID(data_channel_state__port) =  
      _socket_passive_ID(data_channel_state__port'0),  
    all _ID_type'dummy:  
      _valid_active_ID(data_channel_state__port,  
        _ID_type'dummy)  
        = _valid_active_ID(data_channel_state__port'0,  
          _ID_type'dummy),  
    some _event:  
      and (  
        _passive_ID(_event) =  
          _socket_passive_ID(data_channel_state__port),  
        _valid_active_ID(data_channel_state__port,  
          _active_ID(_event)),  
        int'ge(_get_time(_event),  
          _current_time(_external_spec)),  
        _kind(_event) =  
          _event_kind'_SUPER_CMD_GET_KIND,  
        _history(_external_spec, myself)  
          = _apr(_history(_external_spec'0, myself),  
            _event),  
        _return_value(_event) = _err,  
        _state_of_data_channel__get__extract(event)  
          = _state_of_channel__get__type'val(  
            a_data_channelstate, a_data_channel_msg_id)  
      )  
  )  
)
```

```
{ access procedures for select queues pool }
```

```

prog procedure _queues__put_node(
  mvar _external_spec,
  mvar select_queues__port,
  lvar a_node_id,
  lvar a_queue,
  pvar _err) =

  initial (
    _external_spec'0 = _external_spec,
    select_queues__port'0 = select_queues__port
  )

  pre and (
    good(_external_spec),
    _well_sequenced(_external_spec)
  )
  post and (
    _well_sequenced(_external_spec),
    _socket_passive_ID(select_queues__port) =
      _socket_passive_ID(select_queues__port'0),
    all _ID_type'dummy:
      _valid_active_ID(select_queues__port, _ID_type'dummy)
        = _valid_active_ID(select_queues__port'0,
          _ID_type'dummy),
    some _event:
      and (
        _passive_ID(_event) =
          _socket_passive_ID(select_queues__port),
        _valid_active_ID(select_queues__port,
          _active_ID(_event)),
        int'ge(_get_time(_event),
          _current_time(_external_spec)),
        _kind(_event) =
          _event_kind'_SUPER_CMD_GET_KIND,
        _history(_external_spec, myself)
          = _apr(_history(_external_spec'0, myself),
            _event),
        _return_value(_event) = _err,
        _queues__get_or_put__extract(event)
          = _queues__get_or_put__type'val(
            a_node_id, a_queue)
      )
  )

prog procedure _queues__get_head(
  mvar _external_spec,
  mvar select_queue__port,
  pvar a_node_id,
  pvar a_queue,
  pvar _err) =

  initial (

```

```

    _external_spec'0 = _external_spec,
    select_queues__port'0 = select_queues__port
)

pre and (
    good(_external_spec),
    _well_sequenced(_external_spec)
)
post and (
    _well_sequenced(_external_spec),
    _socket_passive_ID(select_queues__port) =
        _socket_passive_ID(select_queues__port'0),
    all _ID_type'dummy:
        _valid_active_ID(select_queues__port, _ID_type'dummy)
            = _valid_active_ID(select_queues__port'0,
                _ID_type'dummy),
    some _event:
        and (
            _passive_ID(_event) =
                _socket_passive_ID(select_queues__port),
            _valid_active_ID(select_queues__port,
                _active_ID(_event)),
            int'ge(_get_time(_event),
                _current_time(_external_spec)),
            _kind(_event) =
                _event_kind' SUPER_CMD_GET_KIND,
            _history(_external_spec, myself)
                = _apr(_history(_external_spec'0, myself),
                    _event),
            _return_value(_event) = _err,
            _queues__get_or_put__extract(event)
                = _queues__get_or_put__type'val(
                    a_node_id, a_queue)
        )
    }
}

prog procedure _queues_remove_node(
    mvar _external_spec,
    mvar select_queues__port,
    lvar a_node_id,
    lvar a_queue,
    pvar _err) =

    initial (
        _external_spec'0 = _external_spec,
        select_queues__port'0 = select_queues__port
    )

    pre and (
        good(_external_spec),
        _well_sequenced(_external_spec)
    )
    post and (
        _well_sequenced(_external_spec),
        _socket_passive_ID(select_queues__port) =
            _socket_passive_ID(select_queues__port'0),
        all _ID_type'dummy:

```

```
    _valid_active_ID(select_queues__port, _ID_type'dummy)
    = _valid_active_ID(select_queues__port'0,
        _ID_type'dummy),
some _event:
    and (
        _passive_ID(_event) =
            _socket_passive_ID(select_queues__port),
        _valid_active_ID(select_queues__port,
            _active_ID(_event)),
        int'ge(_get_time(_event),
            _current_time(_external_spec)),
        _kind(_event) =
            _event_kind'_SUPER_CMD_GET_KIND,
        _history(_external_spec, myself)
            = _apr(_history(_external_spec'0, myself),
                _event),
        _return_value(_event) = _err,
        _queues__get_or_put__extract(event)
            = _queues__get_or_put__type'val(
                a_node_id, a_queue),
    )
)
```

(access procedures for error pool)

```

prog procedure _errors_write(
    mvar _external_spec,
    mvar arb_errors_port,
    lvar an_err_id,
    lvar a_node_id,
    lvar a_queue,
    pvar _err) =

    initial (
        _external_spec'0 = _external_spec,
        arb_errors_port'0 = arb_errors_port
    )

    pre and (
        good(_external_spec),
        _well_sequenced(_external_spec)
    )
    post and (
        _well_sequenced(_external_spec),
        _socket_passive_ID(arb_errors_port) =
            _socket_passive_ID(arb_errors_port'0),
        all _ID_type'dummy:
            _valid_active_ID(arb_errors_port, _ID_type'dummy)
                = _valid_active_ID(arb_errors_port'0,
                    _ID_type'dummy),
        some _event:
            and (
                _passive_ID(_event) =
                    _socket_passive_ID(arb_errors_port),
                _valid_active_ID(arb_errors_port,
                    _active_ID(_event)),
                int'ge(_get_time(_event),
                    _current_time(_external_spec)),
                _kind(_event) =
                    _event_kind'_SUPER_CMD_GET_KIND,
                _history(_external_spec, myself)
                    = _apr(_history(_external_spec'0, myself),
                        _event),
                _return_value(_event) = _err,
                _errors_write_extract(event)
                    = _errors_write_type'val(
                        an_err_id, a_node_id, a_queue)
            )
    )

```

{ Note: type _errors_read_type is declared in main declarations section }

prog var an_err_report: _err_report_type

```

prog procedure _errors_read(
    mvar _external_spec,
    mvar arb_errors_port,

```

```

pvar an_err_report,
pvar _err) =

initial (
  _external_spec'0 = _external_spec,
  arb_errors__port'0 = arb_errors__port
)

pre and (
  good(_external_spec),
  _well_sequenced(_external_spec)
)

post and (
  _well_sequenced(_external_spec),
  _socket_passive_ID(arb_errors__port) =
    _socket_passive_ID(arb_errors__port'0),
  all _ID_type'dummy:
    _valid_active_ID(arb_errors__port, _ID_type'dummy)
      = _valid_active_ID(arb_errors__port'0,
        _ID_type'dummy),
  some _event:
    and (
      _passive_ID(_event) =
        _socket_passive_ID(arb_errors__port),
      _valid_active_ID(arb_errors__port,
        _active_ID(_event)),
      int'ge(_get_time(_event),
        _current_time(_external_spec)),
      _kind(_event) =
        _event_kind'_SUPER_CMD_GET_KIND,
      _history(_external_spec, myself)
        = _apr(_history(_external_spec'0, myself),
          _event),
      _return_value(_event) = _err,
      _errors__read__extract(event)
        = _errors__read__type'val( an_err_report)
    )
)

prog procedure _errors__clear(
  mvar _external_spec,
  mvar arb_errors__port,
  pvar err) =

initial (
  _external_spec'0 = _external_spec,
  arb_errors__port'0 = arb_errors__port
)

pre and (
  good(_external_spec),
  _well_sequenced(_external_spec)
)

post and (
  _well_sequenced(_external_spec),
  _socket_passive_ID(arb_errors__port) =
    _socket_passive_ID(arb_errors__port'0),

```

```
all _ID_type'dummy:
    _valid_active_ID(arb_errors__port, _ID_type'dummy)
        = _valid_active_ID(arb_errors__port'0,
            _ID_type'dummy),
some _event:
    and (
        _passive_ID(_event) =
            _socket_passive_ID(arb_errors__port),
        _valid_active_ID(arb_errors__port,
            _active_ID(_event)),
        int'ge(_get_time(_event),
            _current_time(_external_spec)),
        _kind(_event) =
            _event_kind' SUPER_CMD_GET_KIND,
        _history(_external_spec, myself)
            = _apr(_history(_external_spec'0, myself),
                _event),
        _return_value(_event) = _err
    )
)
```

```
{ access procedures for poll lists pool }
```

```

prog procedure _lists_get_next_to_poll(
  mvar _external_spec,
  mvar poll_lists_port,
  pvar a_node_list,
  pvar a_node_id,
  pvar a_subcycle_count,
  pvar a_primary_list,
  pvar a_secondary_list,
  pvar _err) =

  initial (
    _external_spec'0 = _external_spec,
    poll_lists_port'0 = poll_lists_port
  )

  pre and (
    good(_external_spec),
    _well_sequenced(_external_spec)
  )
  post and (
    _well_sequenced(_external_spec),
    _socket_passive_ID(poll_lists_port) =
      _socket_passive_ID(poll_lists_port'0),
    all _ID_type'dummy:
      _valid_active_ID(poll_lists_port, _ID_type'dummy)
        = _valid_active_ID(poll_lists_port'0,
          _ID_type'dummy),
    some _event:
      and (
        _passive_ID(_event) =
          _socket_passive_ID(poll_lists_port),
        _valid_active_ID(poll_lists_port,
          _active_ID(_event)),
        int'ge(_get_time(_event),
          _current_time(_external_spec)),
        _kind(_event) =
          _event_kind'_SUPER_CMD_GET_KIND,
        _history(_external_spec, myself)
          = _apr(_history(_external_spec'0, myself),
            _event),
        _return_value(_event) = _err,
        _lists_get_next_to_poll_extract(event)
          = _lists_get_next_to_poll_type'val(
            a_node_list, a_node_id,
            a_subcycle_count, a_primary_list,
            a_secondary_list)
      )
  )

prog procedure _lists_reset(
  mvar _external_spec,
  mvar poll_lists_port,
  pvar _err) =

```

```

initial (
    _external_spec'0 = _external_spec,
    poll_lists__port'0 = poll_lists__port
)

pre and (
    good(_external_spec),
    _well_sequenced(_external_spec)
)
post and (
    _well_sequenced(_external_spec),
    _socket_passive_ID(poll_lists__port) =
        _socket_passive_ID(poll_lists__port'0),
    all _ID_type'dummy:
        _valid_active_ID(poll_lists__port, _ID_type'dummy)
            = _valid_active_ID(poll_lists__port'0,
                                _ID_type'dummy),
    some _event:
        and (
            _passive_ID(_event) =
                _socket_passive_ID(poll_lists__port),
            _valid_active_ID(poll_lists__port,
                              _active_ID(_event)),
            int'ge(_get_time(_event),
                    _current_time(_external_spec)),
            _kind(_event) =
                _event_kind'_SUPER_CMD_GET_KIND,
            _history(_external_spec, myself)
                = _apr(_history(_external_spec'0, myself),
                        _event),
            _return_value(_event) = _err
        )
)

```

SHINPADS Specifications

3. Specification of Cmds-to-arb IDA

{ specification of SHINPADS IDA component which contains commands sent from
the supervisor to the arbitrator }

program cmds_to_arb =

environment

var i, j: int

{ channel messages }

type _cmd_kind_type = enum(RESET, ERROR_REQUEST)

prog type _super_cmd_put_type = record
 field command_kind: _cmd_kind_type,
end _super_cmd_put_type

var a_put_cmd: _super_cmd_put_type
function _super_cmd_put_cmd_kind(a_put_cmd): _cmd_kind_type

prog type _super_cmd_get_type = record
 field command_kind: _cmd_kind_type,
end _super_cmd_get_type

var a_get_cmd: _super_cmd_get_type
function _super_cmd_get_cmd_kind(a_get_cmd): _cmd_kind_type

{ id's of components }

type _ID_type = those int'dummy: true

var an_ID: _ID_type

{ ID's of the activities used in this specification }

type _activity_id = enum(
 ARBITRATOR_ID,
 SUPERVISOR_ID
)

{ ID of the IDA used in this specification }

prog const myself: _ID_type

function the_ID(active_id'dummy): _ID_type

{ maps ID onto unique integer }

axiom the_ID_is_one_to_one_axiom(active_id'dummy) =

begin

all active_id'dummy,

active_id'dummy'0:

implies

{

the_ID(active_id'dummy) = the_ID(active_id'dummy'0),

active_id'dummy = active_id'dummy'0

}

end the_ID_is_one_to_one_axiom

{ events }

type _event_return = enum(OK, IDA_FULL, NO_MESSAGE)

```

type _event_kind = enum(
    NIL_EVENT,
    _SUPER_CMD_GET_KIND,
    _SUPER_CMD_PUT_KIND
)

```

```

type _event_type
    var _event: _event_type
    function _kind(_event): _event_kind
    function _time(_event): int
    function _return_value(_event): _event_return
    function _passive_ID(_event): _ID_type
    function _active_ID(_event): _ID_type

```

```

    function _super_cmd_put_extract(_event): _super_cmd_put_type
    function _super_cmd_get_extract(_event): _super_cmd_get_type

```

```

{ event sequences }

```

```

type _event_sequence_type
    var _an_event_sequence: _event_sequence_type
    var _an_event_kind: _event_kind

```

```

{ functions which define a sequence }
    function _is_empty(_an_event_sequence): bool
    function _apr(_an_event_sequence, _event): _event_sequence_type
    function _length(_an_event_sequence): int
    function _right(_an_event_sequence): _event

```

```

{ sequence functions which may be specified in terms of previous functions }
    function _dropr(_an_event_sequence): _event_sequence_type
    function _item(_an_event_sequence, i): _event_type
    function _inrange(_an_event_sequence, i): bool
    function _left(_an_event_sequence): _event_type
    function _dropl(_an_event_sequence): _event_sequence_type

```

```

var _hist : _event_sequence_type
const _empty_sequence: _event_sequence_type

```

```

var state: _event_sequence_type
    { channel state consists of messages written to channel but not
      yet consumed; it is modelled as a sequence of write events }

```

```

{ all events that have occurred in the system }

```

```

prog type IDA_interface_spec
prog var _external_spec: IDA_interface_spec
    function _history(_external_spec, _an_ID): _event_sequence_type
    function _current_time(_external_spec): int
    function _well_sequenced(_external_spec): bool
        { each event on a particular component has a distinct time }

```

```

prog type _socket

```

```

function _socket_passive_ID(_a_socket): _ID_type
function _valid_active_ID(_a_socket, _an_ID): bool

{ error return used by access procedures - network specific }
prog var _err: int

{ windows of IDA }
prog var _super_cmd_put_window: _socket
prog var _super_cmd_get_window: _socket

function modify_state(state, _event): _event_sequence_type =
  { change state of channel to reflect occurrence of event }
  begin
    if
      and (
        _kind(_event) = _event_kind'_SUPER_CMD_PUT_KIND,
        _active_ID(_event) = the_ID(_activity_id'SUPERVISOR_ID),
        _return_value(_event) = OK
      )
    then
      _apr(state, state_event(_event))
      { add the event to the state, changing its kind to STATE }
    else
      if
        and (
          _kind(_event) = _event_kind'_SUPER_CMD_GET_KIND,
          _active_ID(_event) =
            the_ID(_activity_id'ARBITRATOR_ID),
          _return_value(_event) = OK
        )
      then
        _dropl(state)
      else
        state
        { do not change state if read or write was unsuccessful }
      end if
    end if
  end modify_state

function get_state(_hist, state): _event_sequence_type =
  { get state of the channel }
  measure ord'val(_length(_hist))
  begin
    if _hist = _empty_sequence
    then _empty_sequence
    else
      modify_state(get_state(_dropr(_hist), state), _right(_hist))
    end get_state

function derive_state(_hist): _event_sequence_type =
  { get state of channel }
  begin
    get_state(_hist, _empty_sequence)
  end derive_state

const MAXIMUM_WRITE_RATE: int

```

```

function write_seq_ok(_hist): bool =
begin
  { do not write to channel too often }
  all i, j:
    implies
      (
        and (
          _inrange(_hist, i),
          _inrange(_hist, j),
          int'gt(j,i),
          _kind(_item(_hist, i)) =
            _event_kind' SUPER_CMD_PUT_KIND,
          _active_ID(_item(_hist, i)) =
            the_ID(activity_id'SUPERVISOR_ID),
          _kind(_item(_hist, j)) =
            _event_kind' SUPER_CMD_PUT_KIND,
          _active_ID(_item(_hist, i)) =
            the_ID(activity_id'ARBITRATOR_ID)
        ),
        int'ge(
          minus(_time(_item(_hist, j)),
            _time(_item(_hist, i))),
          MAXIMUM_WRITE_RATE)
      )
end write_seq_ok

const MINIMUM_READ_RATE: int

function read_seq_ok(_hist): bool =
begin
  { read channel often enough }
  all i, j:
    implies
      (
        and (
          _inrange(_hist, i),
          _inrange(_hist, j),
          int'ge(
            minus(_time(_item(_hist, j)),
              _time(_item(_hist, i))),
            MINIMUM_READ_RATE)
          ),
        some k:
          and (
            and (
              int'ge(k, i), int'le(k, j)
            ),
            _kind(_item(_hist, k)) =
              _event_kind' SUPER_CMD_GET_KIND,
            _active_ID(_item(_hist, k)) =
              the_ID(activity_id'ARBITRATOR_ID)
          )
        )
end read_seq_ok

function seq_ok(_hist): bool =
begin

```

```

        and (
            write_seq_ok(_hist),
            read_seq_ok(_hist)
        )
    end seq_ok

const IDA_SIZE: int
    ( maximum number of messages in the channel at one time )

function write_append_event_ok(state, _event): bool =
    pre
        and (
            _kind(_event) = _event_kind' SUPER_CMD_PUT_KIND,
            _active_ID(_event) = the_ID(_activity_id' SUPERVISOR_ID)
        )
    begin
        or (
            and (
                _return_value(_event) = _event_return'OK,
                int'lt(_length(state), IDA_SIZE)
            ),
            and (
                _return_value(_event) = _event_return'IDA_FULL,
                int'ge(_length(state), IDA_SIZE)
            )
        )
    end write_append_event_ok

function read_append_event_ok(state, _event): bool =
    pre
        and (
            _kind(_event) = _event_kind' SUPER_CMD_GET_KIND,
            _active_ID(_event) = the_ID(_activity_id' ARBITRATOR_ID)
        )
    begin
        or (
            and (
                _return_value(_event) = _event_return'OK,
                int'gt(_length(state), 0),
                _super_cmd_get_extract(_event) =
                    _super_cmd_get_extract(_left(state))
                ( read message which was first in )
            ),
            and (
                _return_value(_event) = _event_return'NO_MESSAGE,
                _length(state) = 0
            )
        )
    end read_append_event_ok

function append_ok(state, _event): bool =
    begin
        if
            and (
                _kind(_event) = _event_kind' SUPER_CMD_PUT_KIND,
                _active_ID(_event) = the_ID(_activity_id' SUPERVISOR_ID),
            )
        then

```

```

        write_append_event_ok(state, _event)
    else
        if
            and (
                _kind(_event) = _event_kind'_SUPER_CMD_GET_KIND,
                _active_ID(_event) = the_ID(_activity_id'ARBITRATOR_ID)
            )
        then
            read_append_event_ok(state, _event)
        else
            false
        end if
    end if
end append_ok

function ok_sequence(_hist):bool =
    measure ord'val(_length(_hist))
begin
    if _is_empty(sequence)
    then true
    else
        and (
            ok_sequence(_dropr(_hist)),
            seq_ok(_dropr(_hist)),
            int'le(_length(derive_state(_dropr(_hist))), IDA_SIZE),
            append_ok(derive_state(_dropr(_hist)), _right(_hist))
        )
    end if
end ok_sequence

function correct(_hist): bool =
begin
    if _is_empty(_hist)
    then true
    else
        ok_sequence(_hist)
    end if
end correct

function good(_external_spec): bool =
begin
    correct(_history(_external_spec))
end good

{ access procedures for channel from supervisor to arbitrator }

prog var a_command_kind: _cmd_kind_type

prog procedure _super_cmd_put(
    mvar _external_spec,
    mvar _super_cmd_put_window,
    pvar a_command_kind,
    pvar _err) =

```

```

initial (
    _external_spec'0 = _external_spec,
    _super_cmd_put_window'0 = _super_cmd_put_window
)

pre and (
    good(_external_spec),
    _well_sequenced(_external_spec)
)

post and (
    _well_sequenced(_external_spec),
    _socket_passive_ID(_super_cmd_put_window) =
        _socket_passive_ID(_super_cmd_put_window'0),
    all _ID_type'dummy:
        _valid_active_ID(_super_cmd_put_window,
            _ID_type'dummy)
            = _valid_active_ID(_super_cmd_put_window'0,
                _ID_type'dummy),
    some _event:
        and (
            _passive_ID(_event) =
                _socket_passive_ID(_super_cmd_put_window),
            _valid_active_ID(_super_cmd_put_window,
                _active_ID(_event)),
            int'ge(_get_time(_event),
                _current_time(_external_spec)),
            _kind(_event) =
                _event_kind'_SUPER_CMD_GET_KIND,
            _history(_external_spec, myself)
                = _apr(_history(_external_spec'0, myself),
                    _event),
            _return_value(_event) = _err,
            _super_cmd_put_extract(_event)
                = _super_cmd_put_type'val(
                    a_command_kind)
        )
    )

prog procedure _super_cmd_get (
    mvar _external_spec,
    mvar _super_cmd_get_window,
    pvar a_command_kind,
    pvar _err) =

    initial (
        _external_spec'0 = _external_spec,
        _super_cmd_get_window'0 = _super_cmd_get_window
    )

    pre and (
        good(_external_spec),
        _well_sequenced(_external_spec)
    )

    post and (
        _well_sequenced(_external_spec),
        _socket_passive_ID(_super_cmd_get_window) =

```

```

    _socket_passive_ID(_super_cmd_get__window'0),
all _ID_type'dummy:
    _valid_active_ID(_super_cmd_get__window, ID_type'dummy)
        = _valid_active_ID(_super_cmd_get__window'0,
            _ID_type'dummy),
some _event:
    and (
        _passive_ID(_event) =
            _socket_passive_ID(_super_cmd_get__window),
        _valid_active_ID(_super_cmd_get__window,
            _active_ID(_event)),
        int'ge(_get_time(_event),
            _current_time(_external_spec)),
        _kind(_event) =
            _event_kind'_SUPER_CMD_GET_KIND,
        _history(_external_spec, myself)
            = _apr(_history(_external_spec'0, myself),
                _event),
        _return_value(_event) = _err,
        _super_cmd_get__extract(_event)
            = _super_cmd_get__type'val(
                _a_command_kind)
    )
)

```

{ abstract specification of the main program. Note that this must be in the environment section, since every definition outside of the environment must be provided with a model }

```

prog procedure cmds_to_arb(mvar _external_spec,
    mvar _super_cmd_put__window,
    mvar _super_cmd_get__window) =
pre
    _is_empty(_history(_external_spec))
post
    good(_external_spec)

```

end environment

```

main prog procedure cmds_to_arb_spec(mvar _external_spec,
    mvar _super_cmd_put__window,
    mvar _super_cmd_get__window) =
pre
    _is_empty(_history(_external_spec))
post
    good(_external_spec)
begin
    cmds_to_arb_stub(_external_spec, _super_cmdput__window,
        _super_cmd_get__window)
end cmds_to_arb_spec

```

end cmds_to_arb

SHINPADS Specifications

4. High-Level Specification of Bus

{ high-level specifications for bus attached to two servers;
high-level means that the specification deals with events at the windows
of the serversch are attached to activities (rather than at the ports
attached to the bus) }

program bus =

environment

var i, j: int

{ bus messages }

type word_type

prog type _server_get_or_put_type = record
 field first_flag: int,
 field msg_word: word_type
end _server_get_or_put_type

var a_msg: _server_get_or_put_type

function _server_get_or_put_flag(a_msg): int

function _server_get_or_put_word(a_msg): word_type

{ component id's }

type _ID_type = those int'dummy: true

var _an_ID: _ID_type

type passive_id = enum(

 SERVER_A_ID,
 SERVER_B_ID
)

function the_ID(passive_id'dummy): _ID_type

{ maps ID onto unique integer }

axiom the_ID_is_one_to_one_axiom(passive_id'dummy) =
begin

 all passive_id'dummy,
 passive_id'dummy'0:

 implies

 (
 the_ID(passive_id'dummy) =
 the_ID(passive_id'dummy'0),
 passive_id'dummy = passive_id'dummy'0
)

end the_ID_is_one_to_one_axiom

{ events }

type _event_kind = enum(

 NIL_EVENT,
 _SERVER_PUT_KIND,
 _SERVER_GET_KIND
)

type _event_type

var _event: _event_type

```

function _kind(_event): _event_kind
function _passive_ID(_event): _ID_type
function _active_ID(_event): _ID_type
function _time(_event): int
function _return_value(_event): _event_return

function _server_get_or_put_extract(_event): _server_get_or_put_type

{ event sequences }

type _event_sequence_type
var an_event_sequence: _event_sequence_type
var an_event_kind: _event_kind

{ functions which define a sequence }
function _is_empty(an_event_sequence): bool
function _apr(an_event_sequence, _event): _event_sequence_type
function _length(an_event_sequence): int
function _right(an_event_sequence): _event

{ sequence functions which may be specified in terms of previous functions }
function _dropr(an_event_sequence): _event_sequence_type
function _item(an_event_sequence, i): _event_type
function _inrange(an_event_sequence, i): bool
function _left(an_event_sequence): _event_type

function _dropl(an_event_sequence): _event_sequence_type

{ all events that have occurred in the system }
prog type IDA_interface_spec
prog var _external_spec: IDA_interface_spec
function _history(_external_spec, _an_ID): _event_sequence_type
function _current_time(_external_spec): int
function _well_sequenced(_external_spec): bool
    { each event on a particular component has a distinct time }

prog var _err: int

prog type _socket
var _a_socket: _socket
function _socket_passive_ID(_a_socket): passive_id
function _valid_active_ID(_a_socket, _an_ID): bool

prog var server_a_window: _socket
prog var server_b_window: _socket

const A_WHILE: int { time sufficient to get message and broadcast it }
var sourcehist, desthist: _event_sequence_type
var start_time, end_time: int

function server_events(_external_spec, _an_ID): _event_sequence_type =
    { extract sequence of events whose passive event-id is equal to _an_ID }
begin
end server_events

function bus_is_good(_external_spec, start_time, end_time): bool =

```

```

begin
end bus_is_good

function bus_is_good_long_enough(_external_spec, start_time): bool =
begin
    bus_is_good(_external_spec, start_time,
        plus(start_time, A_WHILE))
end bus_is_good_long_enough

function msg_good(_external_spec, sourcehist, desthist): bool =
begin
    all i:
        if
            and (
                _inrange(sourcehist, i),
                _kind(_item(sourcehist,i)) = _SERVER_PUT_KIND,
                _passive_ID(_item(sourcehist,i)) = SERVER_A_ID,
                bus_is_good_long_enough(_external_spec,
                    _time(_item(sourcehist,i)))
            )
        then
            some k:
                and (
                    _inrange(desthist, k),
                    _kind(_item(desthist,k)) = _SERVER_GET_KIND,
                    _passive_ID(_item(desthist,k)) = SERVER_B_ID,
                    _time(_item(desthist,k)) >
                        _time(_item(sourcehist,i)),
                    _time(_item(desthist,k)) <
                        plus(_time(_item(sourcehist,i)), A_WHILE),
                    _server_get_or_put_flag(
                        _server_get_or_put_extract(
                            _item(desthist,k)))
                        = _server_get_or_put_flag(
                            _server_get_or_put_extract(
                                _item(sourcehist,i))),
                    _server_get_or_put_word(
                        _server_get_or_put_extract(
                            _item(desthist,k)))
                        = _server_get_or_put_word(
                            _server_get_or_put_extract(
                                _item(sourcehist,i))),
                )
            else
                true
        end msg_good
    end msg_good

function good(_external_spec): bool =
begin
    msg_good(_external_spec,
        server_events(_external_spec, SERVER_A_ID),
        server_events(_external_spec, SERVER_B_ID))
end good

```

{ access procedures for reading and writing of device }

```

prog var flag: int
prog var word: word_type

prog procedure _server_write(
    mvar _external_spec,
    mvar server_a__window,
    lvar flag,
    lvar word,
    pvar _err) =

    initial (
        _external_spec'0 = _external_spec,
        server_a__window'0 = server_a__window
    )

    pre and (
        good(_external_spec),
        _well_sequenced(_external_spec)
    )
    post and (
        _well_sequenced(_external_spec),
        _socket_passive_ID(server_a__window) =
            _socket_passive_ID(server_a__window'0),
        all _ID_type'dummy:
            _valid_active_ID(server_a__window, _ID_type'dummy)
                = _valid_active_ID(server_a__window'0,
                    _ID_type'dummy),
        some _event:
            and (
                _passive_ID(_event) =
                    _socket_passive_ID(server_a__window),
                _valid_active_ID(server_a__window,
                    _active_ID(_event)),
                int'ge(_get_time(_event),
                    _current_time(_external_spec)),
                _kind(_event) =
                    _event_kind'_SUPER_CMD_GET_KIND,
                _history(_external_spec, myself)
                    = _apr(_history(_external_spec'0, myself),
                        _event),
                _return_value(_event) = _err,
                _server_get_or_put_extract(_event)
                    = _server_get_or_put_type'val(
                        flag, word)
            )
    )

prog procedure _server_read(
    mvar _external_spec,
    mvar server_b__window,
    pvar flag,
    pvar word,
    pvar _err) =

    initial (

```

```

_external_spec'0 = _external_spec,
server_b__window'0 = server_b__window
)

pre and (
    good(_external_spec),
    _well_sequenced(_external_spec)
)
post and (
    _well_sequenced(_external_spec),
    _socket_passive_ID(server_b__window) =
        _socket_passive_ID(server_b__window'0),
    all _ID_type'dummy:
        _valid_active_ID(server_b__window, _ID_type'dummy)
            = _valid_active_ID(server_b__window'0,
                _ID_type'dummy),
    some _event:
        and (
            _passive_ID(_event) =
                _socket_passive_ID(server_b__window),
            _valid_active_ID(server_b__window,
                _active_ID(_event)),
            int'ge(_get_time(_event),
                _current_time(_external_spec)),
            _kind(_event) =
                _event_kind'_SUPER_CMD_GET_KIND,
            _history(_external_spec, myself)
                = _apr(_history(_external_spec'0, myself),
                    _event),
            _return_value(_event) = _err,
            _server__get_or_put__extract(_event)
                = _server__get_or_put__type'val(
                    flag, word)
        )
    )
)

```

{ abstract specification of the main program. Note that this must be in the environment section, since every definition outside of the environment must be provided with a model }

```

prog procedure bus_stub(mvar _external_spec, mvar server_a__window,
    mvar server_b__window) =
    pre
        _is_empty(_history(_external_spec))
    post
        good(_external_spec)

```

end environment

```

main prog procedure bus_spec(mvar _external_spec, mvar server_a__window,
    mvar server_b__window) =
    pre
        _is_empty(_history(_external_spec))
    post
        good(_external_spec)

```

```
begin
    bus_stub(_external_spec, server_a__window, server_b__window)
end bus_spec

end bus
```

SHINPADS Specifications

5. Low-Level Specification of Bus

```
{ low-level specification of bus connected to two servers;
  low-level means that the specification deals with event occurring
  at the ports of the servers which are connected to the bus }
```

```
{ this specification assumes that the bus itself does encoding and decoding
  of electronic signals; it does not attempt to specify how electronic
  signals are mapped into data or vice versa }
```

```
program bus =
```

```
environment
```

```
  var i, j: int
```

```
{ bus messages }
```

```
  prog type bit_type { assume bits sent and received at this level }
  prog type _server_get_or_put_type = record
    field bus_bit: bit_type
  end _server_get_or_put_type
  var a_bus_msg: _server_get_or_put_type
  function _server_get_or_put_bit(a_bus_msg): bit_type
```

```
{ id's of components }
```

```
  type _ID_type = those int'dummy: true
  var _an_ID: _ID_type
  type active_id = enum(
    SERVER_A_ID,
    SERVER_B_ID
  )
```

```
  type passive_id = enum(
    CHANNEL_ID
  )
```

```
  function the_ID(active_id'dummy): _ID_type
    { maps ID onto unique integer }
  axiom the_ID_is_one_to_one_axiom(active_id'dummy) =
    begin
      all active_id'dummy,
        active_id'dummy'0:
          implies
            (
              the_ID(active_id'dummy) =
                the_ID(active_id'dummy'0),
              active_id'dummy = active_id'dummy'0
            )
    end the_ID_is_one_to_one_axiom
```

```
{ events }
```

```
  type event_kind = enum(
    NIL_EVENT,
    _SERVER_PUT_KIND,
    _SERVER_GET_KIND
  )
```

)

type event_type

```

    var _event: _event_type
    function _kind(_event): _event_kind
    function _time(_event): int
    function _return_value(_event): _event_return
    functioniom _passive_ID(_event): _ID_type
    functioniom _active_ID(_event): _ID_type

```

```

    function _server_get_or_put_extract: _server_get_or_put_type

```

{ event sequences }

type _event_sequence_type

```

    var an_event_sequence: _event_sequence_type
    var an_event_kind: _event_kind

```

{ functions which define a sequence }

```

    function _is_empty(an_event_sequence): bool
    function _apr(an_event_sequence, _event): _event_sequence_type
    function _length(an_event_sequence): int
    function _right(an_event_sequence): _event

```

{ sequence functions which may be specified in terms of previous functions }

```

    function _dropr(an_event_sequence): _event_sequence_type
    function _item(an_event_sequence, i): _event_type
    function _inrange(an_event_sequence, i): bool
    function _left(an_event_sequence): _event_type

```

```

    function _dropl(an_event_sequence): _event_sequence_type

```

{ all events that have occurred in the system }

prog type IDA_interface_spec

prog var _external_spec: IDA_interface_spec

```

    function _history(_external_spec, _an_ID): _event_sequence_type
    function _current_time(_external_spec): int
    function _well_sequenced(_external_spec): bool
    { each event on a particular component has a distinct time }

```

prog var _err: int

prog type _socket

```

    var _a_socket: _socket
    function _socket_passive_ID(_a_socket): passive_id
    function _valid_active_ID(_a_socket, _an_ID): bool

```

prog var server_a_port: _socket

prog var server_b_port: _socket

const A_WHILE: int { time sufficient to get message and broadcast it }

var sourcehist, desthist: _event_sequence_type

var start_time, end_time: int

```

function server_events(_external_spec, _an_ID): _event_sequence_type =
  { extract sequence of events whose active event-id is equal to _an_ID }
begin
end server_events

```

```

function bus_is_good(_external_spec, start_time, end_time): bool =
begin
  { 'goodness' of bus is not defined here }
end bus_is_good

```

```

function bus_is_good_long_enough(_external_spec, start_time): bool =
begin
  bus_is_good(_external_spec, start_time,
    plus(start_time, A_WHILE))
end bus_is_good_long_enough

```

```

function msg_good(_external_spec, sourcehist, desthist): bool =
begin
  all i:
    if
      and (
        _inrange(sourcehist, i),
        _kind(_item(sourcehist,i)) = _SERVER_PUT_KIND,
        _active_ID(_item(sourcehist,i)) = SERVER_A_ID,
        bus_is_good_long_enough(_external_spec,
          _time(_item(sourcehist,i)))
      )
    then
      some k:
        and (
          _inrange(desthist, k),
          _kind(_item(desthist,k)) = _SERVER_GET_KIND,
          _active_ID(_item(desthist,k)) = SERVER_B_ID,
          _time(_item(desthist,k)) >
            _time(_item(sourcehist,i)),
          _time(_item(desthist,k)) <
            plus(_time(_item(sourcehist,i)), A_WHILE),
          _server_get_or_put_bit(
            _server_get_or_put_extract(
              _item(desthist,k)))
          = _server_get_or_put_bit(
            _server_get_or_put_extract(
              _item(sourcehist,i)))
        )
      else
        true
    end msg_good
end msg_good

```

```

function good(_external_spec): bool =
begin
  msg_good(_external_spec, server_events(_external_spec,
    SERVER_A_ID),
    server_events(_external_spec, SERVER_B_ID))
end good

```

```
{ access procedures for channel from supervisor to arbitrator }
```

```
prog var a_bit: bit_type
```

```
prog procedure _server_write(
```

```
    mvar _external_spec,  
    mvar server_a__port,  
    pvar a_bit,  
    pvar _err) =
```

```
    initial (
```

```
        _external_spec'0 = _external_spec,  
        server_a__port'0 = server_a__port  
    )
```

```
    pre and (
```

```
        good(_external_spec),  
        _well_sequenced(_external_spec)
```

```
    )
```

```
    post and (
```

```
        _well_sequenced(_external_spec),  
        _socket_passive_ID(server_a__port) =  
        _socket_passive_ID(server_a__port'0),  
        all _ID_type'dummy:  
            _valid_active_ID(server_a__port, _ID_type'dummy)  
            = _valid_active_ID(server_a__port'0,  
                _ID_type'dummy),
```

```
    some _event:
```

```
        and (
```

```
            _passive_ID(_event) =  
            _socket_passive_ID(server_a__port),  
            _valid_active_ID(server_a__port,  
                _active_ID(_event)),  
            int'ge(_get_time(_event),  
                _current_time(_external_spec)),  
            _kind(_event) =  
            _event_kind'_SUPER_CMD_GET_KIND,  
            _history(_external_spec, myself)  
            = _apr(_history(_external_spec'0, myself),  
                _event),  
            _return_value(_event) = _err,  
            _server_get_or_put_extract(event)  
            = get_from_super_type'val(a_bit)
```

```
        )
```

```
    )
```

```
prog procedure _server_read(
```

```
    mvar _external_spec,  
    mvar server_b__port,  
    pvar a_bit,  
    pvar _err) =
```

```
    initial (
```

```
        _external_spec'0 = _external_spec,
```

```

server_b__port'0 = server_b__port
)

pre and (
    good(_external_spec),
    _well_sequenced(_external_spec)
)

post and (
    _well_sequenced(_external_spec),
    _socket_passive_ID(server_b__port) =
        _socket_passive_ID(server_b__port'0),
    all _ID_type'dummy:
        _valid_active_ID(server_b__port, _ID_type'dummy)
            = _valid_active_ID(server_b__port'0,
                _ID_type'dummy),
    some _event:
        and (
            _passive_ID(_event) =
                _socket_passive_ID(server_b__port),
            _valid_active_ID(server_b__port,
                _active_ID(_event)),
            int'ge(_get_time(_event),
                _current_time(_external_spec)),
            _kind(_event) =
                _event_kind'_SUPER_CMD_GET_KIND,
            _history(_external_spec, myself)
                = _apr(_history(_external_spec'0, myself),
                    _event),
            _return_value(_event) = _err,
            _server_get_or_put_extract(event)
                = get_from_super_type'val(a_bit)
        )
    )
)

```

{ abstract specification of the main program. Note that this must be in the environment section, since every definition outside of the environment must be provided with a model }

```

prog procedure bus_stub(mvar _external_spec, mvar server_a__port,
    mvar server_b__port) =
    pre
        _is_empty(_history(_external_spec))
    post
        good(_external_spec)

```

end environment

```

main prog procedure bus_spec(mvar _external_spec, mvar server_a__port,
    mvar server_b__port) =
    pre
        _is_empty(_history(_external_spec))
    post
        good(_external_spec)
    begin
        bus_stub(_external_spec, server_a__port, server_b__port)
    end bus_spec

```

BUSLOW

Page 6

end bus

References

- [1] Joint IECCA and MUF Committee on Mascot (JIMCOM)
The Official Handbook of Mascot Version 3.1
Royal Signals and Radar Establishment, St. Andrews Road, Malvern,
Worcestershire WR14 3PS, U. K.
1987

- [2] Higginson, M. M. and Andrews, J. David
Distributed System Design Using Mascot and m-Verdi
Real-time Distributed Systems
Andyne Computing Limited
June 1988

- [3] SPERRY-UNIVAC
Serial Data Bus System User's Manual for the Shinpads Engineering Development
1982

- [4] Craigen, D.
A Description of m-Verdi
EVES Technical Report, TR-87-5420,
I.P. Sharp Associates, Limited
265 Carling Avenue, Suite 600
Ottawa, Ontario, Canada
November, 1987

